

SAMS  
**Teach  
Yourself**

- 全球销量逾百万册的系列图书
- 连续十余年打造的经典品牌
- 直观、循序渐进的学习教程
- 掌握关键知识的最佳起点
- 秉承Read Less, Do More（精读多练）的教学理念
- 以示例引导读者完成最常见的任务

每章内容针对初学者精心设计，**1**小时轻松阅读学习，  
**24**小时彻底掌握关键知识

每章**案例与练习题**助你轻松完成常见任务，  
通过**实践**提高应用技能，巩固所学知识

# SQL

## 入门经典（第5版）

Ryan Stephens  
[美] Ron Plew 著  
Arie D. Jones  
井中月 郝记生 译

SQL入门经典（第5版）

[美]Ryan Stephens Ron Plew Arie D.Jones 着  
井中月 郝记生 译

人民邮电出版社

北京

## 内容提要

本书的作者都是数据库教学与应用的专家，有着丰富的经验。本书详细介绍了SQL语言的基本语法、基本概念，说明了各种SQL实现与ANSI标准之间的差别。书中包含了大量的范例，直观地说明了如何使用SQL对数据进行处理。每章后面还有针对性很强的测验与练习，能够帮助读者更好地理解 and 掌握学习的内容。在最后的附录里还有关于安装MySQL 的详细介绍、书中用到的关键SQL语句、测验和练习的答案。

本书的内容层次清晰，针对性强，非常适合初学者作为入门教材。

## [关于作者](#)

本书的作者们10多年来研究、应用和总结了SQL标准以及这些标准在关系型数据库的应用。

**Ryan Stephens**和**Ron Plew**是Perpetual Technologies (PTI) 公司的老板、发言人和共同创建者，这是一家正在高速发展的IT管理与咨询公司，专门从事数据库技术，特别是Oracle和SQL服务程序在各种UNIX、Linux和Windows平台上的运行。Ryan和Ron最初从事数据分析和数据库管理，现在领导着一个专家小组，为全世界范围内的客户管理数据库。他们还在印第安纳波利斯的 Indiana University-Purdue大学创办并教授数据库课程达5年之久，并且编写了10余本关于Oracle、SQL、数据库设计和重要系统高可用性方面的图书。

**Arie D. Jones**是PTI公司的高级SQL Server数据库管理员和分析员，领导着一个专家小组负责数据库环境与应用程序的规划、设计、开发、部署，从而让每个客户都获得最佳的工具与服务的组合。他是技术事件的定期发言人，并且编写了多本关于数据库的图书和论文。



# 目 录

---

[封面](#)

[扉页](#)

[内容提要](#)

[关于作者](#)

[第一部分 SQL概念综述](#)

[第1章 欢迎来到SQL世界](#)

[1.1 SQL定义及历史](#)

[1.1.1 什么是SQL](#)

[1.1.2 什么是ANSI SQL](#)

[1.1.3 新标准：SQL-2008](#)

[1.1.4 什么是数据库](#)

[1.1.5 关系型数据库](#)

[1.1.6 客户端/服务器技术](#)

[1.1.7 基于Web的数据库系统](#)

### [1.1.8 主流数据库厂商](#)

## [1.2 SQL会话](#)

### [1.2.1 CONNECT](#)

### [1.2.2 DISCONNECT和EXIT](#)

## [1.3 SQL命令的类型](#)

### [1.3.1 定义数据库结构](#)

### [1.3.2 操作数据](#)

### [1.3.3 选择数据](#)

### [1.3.4 数据控制语言](#)

### [1.3.5 数据管理命令](#)

### [1.3.6 事务控制命令](#)

## [1.4 本书使用的数据库](#)

### [1.4.1 表命名标准](#)

### [1.4.2 数据一瞥](#)

### [1.4.3 表的构成](#)

### [1.4.4 范例和练习](#)

## [1.5 小结](#)

## [1.6 问与答](#)

## [1.7 实践](#)

### [1.7.1 测验](#)

### [1.7.2 练习](#)

## [第二部分 创建数据库](#)

### [第2章 定义数据结构](#)

## [2.1 数据是什么](#)

## [2.2 基本数据类型](#)

### [2.2.1 定长字符串](#)

### [2.2.2 变长字符串](#)

### [2.2.3 大对象类型](#)

### [2.2.4 数值类型](#)

### [2.2.5 小数类型](#)

### [2.2.6 整数](#)

### [2.2.7 浮点数](#)

### [2.2.8 日期和时间类型](#)

[2.2.9 直义字符串](#)

[2.2.10 NULL数据类型](#)

[2.2.11 布尔值](#)

[2.2.12 自定义类型](#)

[2.2.13 域](#)

[2.3 小结](#)

[2.4 问与答](#)

[2.5 实践](#)

[2.5.1 测验](#)

[2.5.2 练习](#)

[第3章 管理数据库对象](#)

[3.1 什么是数据库对象](#)

[3.2 什么是规划](#)

[3.3 表：数据的主要存储方式](#)

[3.3.1 列](#)

[3.3.2 行](#)

[3.3.3 CREATE TABLE语句](#)

### [3.3.4 命名规范](#)

### [3.3.5 ALTER TABLE命令](#)

### [3.3.6 从现有表新建另一个表](#)

### [3.3.7 删除表](#)

## [3.4 完整性约束](#)

### [3.4.1 主键约束](#)

### [3.4.2 唯一性约束](#)

### [3.4.3 外键约束](#)

### [3.4.4 NOT NULL约束](#)

### [3.4.5 检查约束](#)

### [3.4.6 去除约束](#)

## [3.5 小结](#)

## [3.6 问与答](#)

## [3.7 实践](#)

### [3.7.1 测验](#)

### [3.7.2 练习](#)

## 第4章 规格化过程

### 4.1 规格化数据库

#### 4.1.1 原始数据库

#### 4.1.2 数据库逻辑设计

#### 4.1.3 规格形式

#### 4.1.4 命名规范

#### 4.1.5 规格化的优点

#### 4.1.6 规格化的缺点

### 4.2 去规格化数据库

### 4.3 小结

### 4.4 问与答

### 4.5 实践

#### 4.5.1 测验

#### 4.5.2 练习

## 第5章 操作数据

### 5.1 数据操作概述

### 5.2 用新数据填充表

### [5.2.1 把数据插入到表](#)

### [5.2.2 给表里指定列插入数据](#)

### [5.2.3 从另一个表插入数据](#)

### [5.2.4 插入NULL值](#)

## [5.3 更新现有数据](#)

### [5.3.1 更新一系列的数据](#)

### [5.3.2 更新一条或多记录里的多个字段](#)

## [5.4 从表里删除数据](#)

## [5.5 小结](#)

## [5.6 问与答](#)

## [5.7 实践](#)

### [5.7.1 测验](#)

### [5.7.2 练习](#)

# [第6章 管理数据库事务](#)

## [6.1 什么是事务](#)

## [6.2 控制事务](#)

[6.2.1 COMMIT命令](#)

[6.2.2 ROLLBACK命令](#)

[6.2.3 SAVEPOINT命令](#)

[6.2.4 ROLLBACK TO SAVEPOINT命令](#)

[6.2.5 RELEASE SAVEPOINT命令](#)

[6.2.6 SET TRANSACTION命令](#)

[6.3 事务控制与数据库性能](#)

[6.4 小结](#)

[6.5 问与答](#)

[6.6 实践](#)

[6.6.1 测验](#)

[6.6.2 练习](#)

[第三部分 从查询中获得有效的结果](#)

[第7章 数据库查询](#)

[7.1 什么是查询](#)

[7.2 SELECT语句](#)

[7.2.1 SELECT语句](#)



### [7.2.2 FROM子句](#)

### [7.2.3 WHERE子句](#)

### [7.2.4 ORDER BY子句](#)

### [7.2.5 大小写敏感性](#)

## [7.3 简单查询的范例](#)

### [7.3.1 统计表里的记录数量](#)

### [7.3.2 从另一个用户表里选择数据](#)

### [7.3.3 使用字段别名](#)

## [7.4 小结](#)

## [7.5 问与答](#)

## [7.6 实践](#)

### [7.6.1 测验](#)

### [7.6.2 练习](#)

## [第8章 使用操作符对数据进行分类](#)

### [8.1 什么是SQL里的操作符](#)

### [8.2 比较操作符](#)

[8.2.1 相等](#)

[8.2.2 不等于](#)

[8.2.3 小于和大于](#)

[8.2.4 比较操作符的组合](#)

[8.3 逻辑操作符](#)

[8.3.1 IS NULL](#)

[8.3.2 BETWEEN](#)

[8.3.3 IN](#)

[8.3.4 LIKE](#)

[8.3.5 EXISTS](#)

[8.3.6 ALL、SOME和ANY操作符](#)

[8.4 连接操作符](#)

[8.4.1 AND](#)

[8.4.2 OR](#)

[8.5 求反操作符](#)

[8.5.1 不相等](#)

[8.5.2 NOT BETWEEN](#)

[8.5.3 NOT IN](#)

[8.5.4 NOT LIKE](#)

[8.5.5 IS NOT NULL](#)

[8.5.6 NOT EXISTS](#)

[8.6 算术操作符](#)

[8.6.1 加法](#)

[8.6.2 减法](#)

[8.6.3 乘法](#)

[8.6.4 除法](#)

[8.6.5 算术操作符的组合](#)

[8.7 小结](#)

[8.8 问与答](#)

[8.9 实践](#)

[8.9.1 测验](#)

[8.9.2 练习](#)

[第9章 汇总查询得到的数据](#)

## [9.1 什么是汇总函数](#)

### [9.1.1 COUNT函数](#)

### [9.1.2 SUM函数](#)

### [9.1.3 AVG函数](#)

### [9.1.4 MAX函数](#)

### [9.1.5 MIN函数](#)

## [9.2 小结](#)

## [9.3 问与答](#)

## [9.4 实践](#)

### [9.4.1 测验](#)

### [9.4.2 练习](#)

## [第10章 数据排序与分组](#)

### [10.1 为什么要对数据进行分组](#)

## [10.2 GROUP BY子句](#)

### [10.2.1 分组函数](#)

### [10.2.2 对选中的数据进行分组](#)

### [10.2.3 创建分组和使用汇总函数](#)

#### [10.2.4 以整数代表字段名称](#)

### [10.3 GROUP BY与ORDER BY](#)

#### [10.4 CUBE和ROLLUP语句](#)

#### [10.5 HAVING子句](#)

#### [10.6 小结](#)

#### [10.7 问与答](#)

### [10.8 实践](#)

#### [10.8.1 测验](#)

#### [10.8.2 练习](#)

## [第11章 调整数据的外观](#)

### [11.1 ANSI字符函数](#)

### [11.2 常用字符函数](#)

#### [11.2.1 串接函数](#)

#### [11.2.2 TRANSLATE函数](#)

#### [11.2.3 REPLACE](#)

#### [11.2.4 UPPER](#)

[11.2.5 LOWER](#)

[11.2.6 SUBSTR](#)

[11.2.7 INSTR](#)

[11.2.8 LTRIM](#)

[11.2.9 RTRIM](#)

[11.2.10 DECODE](#)

[11.3 其他字符函数](#)

[11.3.1 LENGTH](#)

[11.3.2 IFNULL（检查NULL值）](#)

[11.3.3 COALESCE](#)

[11.3.4 LPAD](#)

[11.3.5 RPAD](#)

[11.3.6 ASCII](#)

[11.4 算术函数](#)

[11.5 转换函数](#)

[11.5.1 字符串转换为数字](#)

[11.5.2 数字转换为字符串](#)

## [11.6 字符函数的组合使用](#)

## [11.7 小结](#)

## [11.8 问与答](#)

## [11.9 实践](#)

### [11.9.1 测验](#)

### [11.9.2 练习](#)

## [第12章 日期和时间](#)

## [12.1 日期是如何存储的](#)

### [12.1.1 日期和时间的标准数据类型](#)

### [12.1.2 DATETIME元素](#)

### [12.1.3 不同实现的日期类型](#)

## [12.2 日期函数](#)

### [12.2.1 当前日期](#)

### [12.2.2 时区](#)

### [12.2.3 时间与日期相加](#)

### [12.2.4 其他日期函数](#)

## [12.3 日期转换](#)

### [12.3.1 日期描述](#)

### [12.3.2 日期转换为字符串](#)

### [12.3.3 字符串转换为日期](#)

## [12.4 小结](#)

## [12.5 问与答](#)

## [12.6 实践](#)

### [12.6.1 测验](#)

### [12.6.2 练习](#)

## [第四部分 创建复杂的数据库查询](#)

## [第13章 在查询里结合表](#)

### [13.1 从多个表获取数据](#)

### [13.2 结合的类型](#)

#### [13.2.1 结合条件的位置](#)

#### [13.2.2 等值结合](#)

#### [13.2.3 使用表的别名](#)

#### [13.2.4 不等值结合](#)



#### [13.2.5 外部结合](#)

#### [13.2.6 自结合](#)

#### [13.2.7 结合多个主键](#)

### [13.3 需要考虑的事项](#)

#### [13.3.1 使用基表](#)

#### [13.3.2 笛卡尔积](#)

### [13.4 小结](#)

### [13.5 问与答](#)

### [13.6 实践](#)

#### [13.6.1 测验](#)

#### [13.6.2 练习](#)

## [第14章 使用子查询定义未确定数据](#)

### [14.1 什么是子查询](#)

#### [14.1.1 子查询与SELECT语句](#)

#### [14.1.2 子查询与INSERT语句](#)

#### [14.1.3 子查询与UPDATE语句](#)

#### [14.1.4 子查询与DELETE语句](#)

#### [14.2 嵌套的子查询](#)

#### [14.3 关联子查询](#)

#### [14.4 子查询的效率](#)

#### [14.5 小结](#)

#### [14.6 问与答](#)

#### [14.7 实践](#)

##### [14.7.1 测验](#)

##### [14.7.2 练习](#)

### [第15章 组合多个查询](#)

#### [15.1 单查询与组合查询](#)

#### [15.2 组合查询操作符](#)

##### [15.2.1 UNION](#)

##### [15.2.2 UNION ALL](#)

##### [15.2.3 INTERSECT](#)

##### [15.2.4 EXCEPT](#)

#### [15.3 组合查询里使用ORDER BY](#)

## [15.4 组合查询里使用GROUP BY](#)

## [15.5 获取准确的数据](#)

## [15.6 小结](#)

## [15.7 问与答](#)

## [15.8 实践](#)

### [15.8.1 测验](#)

### [15.8.2 练习](#)

## [第五部分 SQL性能调整](#)

## [第16章 利用索引改善性能](#)

## [16.1 什么是索引](#)

## [16.2 索引是如何工作的](#)

## [16.3 CREATE INDEX命令](#)

## [16.4 索引的类型](#)

### [16.4.1 单字段索引](#)

### [16.4.2 唯一索引](#)

### [16.4.3 组合索引](#)

#### [16.4.4 隐含索引](#)

#### [16.5 何时考虑使用索引](#)

#### [16.6 何时应该避免使用索引](#)

#### [16.7 修改索引](#)

#### [16.8 删除索引](#)

#### [16.9 小结](#)

#### [16.10 问与答](#)

#### [16.11 实践](#)

##### [16.11.1 测验](#)

##### [16.11.2 练习](#)

### [第17章 改善数据库性能](#)

#### [17.1 什么是SQL语句调整](#)

#### [17.2 数据库调整与SQL语句调整](#)

#### [17.3 格式化SQL语句](#)

##### [17.3.1 为提高可读性格式化SQL语句](#)

##### [17.3.2 FROM子句里的表](#)

##### [17.3.3 结合条件的次序](#)

#### [17.3.4 最严格条件](#)

### [17.4 全表扫描](#)

### [17.5 其他性能考虑](#)

#### [17.5.1 使用LIKE操作符和通配符](#)

#### [17.5.2 避免使用OR操作符](#)

#### [17.5.3 避免使用HAVING子句](#)

#### [17.5.4 避免大规模排序操作](#)

#### [17.5.5 使用存储过程](#)

#### [17.5.6 在批加载时关闭索引](#)

### [17.6 基于成本的优化](#)

### [17.7 性能工具](#)

### [17.8 小结](#)

### [17.9 问与答](#)

### [17.10 实践](#)

#### [17.10.1 测验](#)

#### [17.10.2 练习](#)

## 第六部分 使用SQL管理用户和安全

### 第18章 管理数据库用户

#### 18.1 数据库的用户管理

##### 18.1.1 用户的类型

##### 18.1.2 谁管理用户

##### 18.1.3 用户在数据库里的位置

##### 18.1.4 不同规划里的用户

#### 18.2 管理过程

##### 18.2.1 创建用户

##### 18.2.2 创建规划

##### 18.2.3 删除规划

##### 18.2.4 调整用户

##### 18.2.5 用户会话

##### 18.2.6 禁止用户访问

#### 18.3 数据库用户使用的工具

#### 18.4 小结

#### 18.5 问与答

## [18.6 实践](#)

### [18.6.1 测验](#)

### [18.6.2 练习](#)

## [第19章 管理数据库安全](#)

### [19.1 什么是数据库安全](#)

### [19.2 什么是权限](#)

#### [19.2.1 系统权限](#)

#### [19.2.2 对象权限](#)

#### [19.2.3 谁负责授予和撤销权限](#)

### [19.3 控制用户访问](#)

#### [19.3.1 GRANT命令](#)

#### [19.3.2 REVOKE命令](#)

#### [19.3.3 控制对单独字段的访问](#)

#### [19.3.4 数据库账户PUBLIC](#)

#### [19.3.5 权限组](#)

### [19.4 通过角色控制权限](#)

#### [19.4.1 CREATE ROLE语句](#)

#### [19.4.2 DROP ROLE语句](#)

#### [19.4.3 SET ROLE语句](#)

### [19.5 小结](#)

### [19.6 问与答](#)

### [19.7 实践](#)

#### [19.7.1 测验](#)

#### [19.7.2 练习](#)

## [第七部分 摘要数据结构](#)

### [第20章 创建和使用视图及异名](#)

#### [20.1 什么是视图](#)

##### [20.1.1 使用视图来简化数据访问](#)

##### [20.1.2 使用视图作为一种安全角式](#)

##### [20.1.3 使用视图维护摘要数据](#)

#### [20.2 创建视图](#)

##### [20.2.1 从一个表创建视图](#)

##### [20.2.2 从多个表创建视图](#)



### [20.2.3 从视图创建视图](#)

## [20.3 WITH CHECK OPTION](#)

### [20.4 从视图创建表](#)

### [20.5 视图与ORDER BY子句](#)

### [20.6 通过视图更新数据](#)

### [20.7 删除视图](#)

### [20.8 嵌套视图对性能的影响](#)

## [20.9 什么是异名](#)

### [20.9.1 创建异名](#)

### [20.9.2 删除异名](#)

## [20.10 小结](#)

## [20.11 问与答](#)

## [20.12 实践](#)

### [20.12.1 测验](#)

### [20.12.2 练习](#)

## [第21章 使用系统目录](#)

## [21.1 什么是系统目录](#)

## [21.2 如何创建系统目录](#)

## [21.3 系统目录里包含什么内容](#)

### [21.3.1 用户数据](#)

### [21.3.2 安全信息](#)

### [21.3.3 数据库设计信息](#)

### [21.3.4 性能统计](#)

## [21.4 不同实现里的系统目录表格](#)

## [21.5 查询系统目录](#)

## [21.6 更新系统目录对象](#)

## [21.7 小结](#)

## [21.8 问与答](#)

## [21.9 实践](#)

### [21.9.1 测验](#)

### [21.9.2 练习](#)

## [第八部分 在实际工作中应用SQL知识](#)

## [第22章 高级SQL主题](#)

## [22.1 光标](#)

### [22.1.1 打开光标](#)

### [22.1.2 从光标获取数据](#)

### [22.1.3 关闭光标](#)

## [22.2 存储过程和函数](#)

## [22.3 触发器](#)

### [22.3.1 CREATE TRIGGER语句](#)

### [22.3.2 DROP TRIGGER语句](#)

### [22.3.3 FOR EACH ROW语句](#)

## [22.4 动态SQL](#)

## [22.5 调用级接口](#)

## [22.6 使用SQL生成SQL](#)

## [22.7 直接SQL与嵌入SQL](#)

## [22.8 窗口表格函数](#)

## [22.9 使用XML](#)

## [22.10 小结](#)

## [22.11 问与答](#)

## [22.12 实践](#)

### [22.12.1 测验](#)

### [22.12.2 练习](#)

## [第23章 SQL扩展到企业、互联网和内部网](#)

## [23.1 SQL与企业](#)

### [23.1.1 后台程序](#)

### [23.1.2 前台程序](#)

## [23.2 访问远程数据库](#)

### [23.2.1 ODBC](#)

### [23.2.2 JDBC](#)

### [23.2.3 OLE DB](#)

### [23.2.4 厂商连接产品](#)

### [23.2.5 通过Web接口访问远程数据库](#)

## [23.3 SQL与互联网](#)

### [23.3.1 让数据可以被全世界的顾客使用](#)

### [23.3.2 向雇员和授权顾客提供数据](#)

## [23.4 SQL与内部网](#)

## [23.5 小结](#)

## [23.6 问与答](#)

## [23.7 实践](#)

### [23.7.1 测验](#)

### [23.7.2 练习](#)

## [第24章 标准SQL的扩展](#)

## [24.1 各种实现](#)

### [24.1.1 不同实现之间的区别](#)

### [24.1.2 遵循ANSI SQL](#)

### [24.1.3 SQL的扩展](#)

## [24.2 扩展范例](#)

### [24.2.1 Transact-SQL](#)

### [24.2.2 PL/SQL](#)

### [24.2.3 MySQL](#)

## [24.3 交互SQL语句](#)

## [24.4 小结](#)

## [24.5 问与答](#)

## [24.6 实践](#)

### [24.6.1 测验](#)

### [24.6.2 练习](#)

## [第九部分 附录](#)

### [附录A 常用SQL命令](#)

#### [A.1 SQL语句](#)

#### [A.2 SQL子句](#)

### [附录B 使用数据库进行练习](#)

#### [B.1 在Windows操作系统中安装MySQL的指令](#)

#### [B.2 在Windows操作系统中安装Oracle的指令](#)

#### [B.3 在Windows操作系统中安装Microsoft SQL Server的指令](#)

### [附录C 测验和练习的答案](#)

### [附录D 本书范例的CREATE TABLE语句](#)

#### [D.1 MySQL](#)

## [D.2 Oracle和SQL Server](#)

### [附录E 书中范例所涉数据的INSERT语句](#)

#### [E.1 MySQL和SQL Server](#)

##### [E.1.1 EMPLOYEE\\_TBL](#)

##### [E.1.2 EMPLOYEE\\_PAY\\_TBL](#)

##### [E.1.3 CUSTOMER\\_TBL](#)

##### [E.1.4 ORDERS\\_TBL](#)

##### [E.1.5 PRODUCTS\\_TBL](#)

#### [E.2 Oracle](#)

##### [E.2.1 EMPLOYEE\\_TBL](#)

##### [E.2.2 EMPLOYEE\\_PAY\\_TBL](#)

##### [E.2.3 CUSTOMER\\_TBL](#)

##### [E.2.4 ORDERS\\_TBL](#)

##### [E.2.5 PRODUCTS\\_TBL](#)

### [附录F 额外练习](#)

### [术语表](#)

### [其他](#)

版权



## 第一部分 SQL概念综述

### 第1章 欢迎来到SQL世界

#### 第1章 欢迎来到SQL世界

本章的重点包括：

SQL历史简介

介绍数据库管理系统

一些基本术语和概念

介绍本书所使用的数据库

欢迎来到SQL的世界，体验当今世界庞大的不断发展的数据库技术。通过阅读本书，我们可以获得很多的知识，而这些是在当今关系型数据库和数据管理领域生存所必需的。由于首先必须要介绍SQL的背景知识和一些预备知识，本章的主要内容是对后续章节的概述，这显得有些单调，但这些貌似无聊的内容却是体会本书后续精彩内容的基础。

#### 1.1 SQL定义及历史

当今时代的任何事务都涉及数据，人们需要使用某种有组织的方法或机制来管理和检索数据。如果数据被保存在数据库中，这种机制便被称为数据库管理系统（DBMS）。数据库管理系统已经产生多年了，其中大多数源自于大型机上的平面文件系统。随着技术的发展，在不断增长的商业需要、不断增加的共享数据和互联网的推动下，数据库管理系统的使用已经偏离了其原始方向。

信息管理的现代浪潮主要是由关系型数据库管理系统（RDBMS）实现的，后者是从传统DBMS派生出来的。

现代数据库与客户端/服务器或Web技术相结合在当今是很常见的模式，公司使用这些方式来管理数据，从而在相应的市场保持竞争力。很多公司的趋势是从客户端/服务器模式转移到Web模式，从而避免用户在访问重要数据时受到地点的限制。下面几个小节将讨论SQL和关系型数据库，后者是当今最通用的DBMS实现。很好地理解关系型数据库，以及如何在当今信息技术世界利用SQL来管理数据，对于理解SQL语言是十分重要的。

### 1.1.1 什么是SQL

“结构化查询语言（SQL）”是与关系型数据库进行通信的标准语言，最初是由IBM公司以E.F. Codd博士的论文《A Relational Model of Data for Large Shared Data Banks》为原型开发出来的。在之后不久的1979年，Relational Software公司（后来更名为Oracle公司）发布了第一个SQL产品：ORACLE，现在已经成为关系型数据库技术的领军者。

当我们去别的国家旅行时，需要了解其语言才能更加方便。举例来说，如果服务员只能使用其本国语言，那我们用母语点菜可能就会有麻烦。如果把数据库看作一个要从中进行信息搜索的外国，那么SQL就是我们向数据库表达需求的语言，我们可以利用SQL进行查询，从数据库里获得特定的信息。

### 1.1.2 什么是ANSI SQL

“美国国家标准化组织（ANSI）”是一个核准多种行业标准的组织。SQL作为关系型数据库所使用的标准语言，最初是基于IBM的实现在1986年被批准的。1987年，“国际标准化组织（ISO）”把ANSI SQL作为国际标准。这个标准在1992年进行了修订（SQL-92），1999年再次修订（SQL-99）。目前最新的标准是2008年7月开始采用的SQL-2008。

### 1.1.3 新标准：SQL-2008

SQL-2008由9个相关的文档组成，在不远的将来还可能增加其他文档，以扩展标准来适应新出现的技术。

第1部分——SQL/架构：指定实现一致性的一般性需求，定义SQL的基本概念。

第2部分——SQL/基础：定义SQL的语法和操作。

第3部分——SQL/调用级接口：定义程序编程与SQL的接口。

第4部分——SQL/持久存储模块：定义控制结构，进而定义SQL例程。还定义了包含SQL例程的模块。

第9部分——外部数据管理（SQL/MED）：定义SQL的扩展，用于通过使用数据包裹支持外部数据管理；还定义了数据链类型。

第10部分——对象语言绑定：定义SQL的扩展，支持把SQL语句内嵌到用Java编写的程序。

第11部分——信息和定义方案：定义信息方案和定义方案的规范，提供与SQL数据相关的结构和安全信息。

第13部分——使用Java编程语言的例程和类型：定义以SQL例程形式调用Java静态例程和类的功能。

第14部分——XML相关规范：定义SQL使用XML的方式。

对于新的ANSI标准（SQL-2008），DBMS声称的兼容有两个级别：核心SQL支持和增强SQL支持。在下面这个网页上可以找到ANSI SQL标准的超级链接：[www.informit.com/title/9780672335419](http://www.informit.com/title/9780672335419)。

ANSI表示“美国国家标准化组织”，负责规划各种产品和概念的标准。

标准显然是有好处的，当然有时也有不足之处。最重要的是，标准指引厂商沿着恰当的开发方向前进。就SQL来说，标准提供了必要基本原则的骨架，从而最终让不同的实现之间保持一致性，更好地实现可移植性（不仅是对于数据库编程，而且对于数据库整体和管理数据库的个人而言）。

有人认为标准并不是那么好，它限制了灵活性和特定实现的功能。然而，大多数遵循标准的厂商都在特定产品里实现了对标准SQL的增强，从而弥补了这种问题。

综合考虑正反两方面的因素，标准还是好的。标准定义了在任何SQL完整实现中都应该具有的功能，规划的基本概念不仅让各种相互竞争的SQL实现保持一致性，也提高了SQL程序员的价值。

所谓SQL实现是指特定厂商的SQL产品或关系型数据库管理系统。需要说明的是，SQL实现之间的差别是很大的。虽然有些实现的大部分是与ANSI兼容的，但没有任何一种实现完全遵循标准。另外，ANSI标准里为了保持兼容性而必须遵守的功能列表在近些年并没有太大改变，因此，新版本的RDBMS也必将保持与ANSI SQL的兼容性。

#### 1.1.4 什么是数据库

简单来说，数据库就是数据集合。我们可以把数据库看成这样一种有组织的机制：它能够存储信息，用户能够以有效且高效的方式检索其中的信息。

事实上，人们每天都在使用数据库，只是没有察觉到。电话簿就是个数据库，其中的数据包括个人的姓名、地址和电话号码。这些数据是按字母排序或是索引排序的，让用户能够方便地找到特定的本地居民。实际上，这些数据保存在计算机上的某个数据库里。毕竟这些电话簿的每一页都不是手写的，而且每年都会发布一个新版本。

数据库必须被维护。由于居民会搬到其他城市或州，电话簿里的项目就需要删除或添加。类似地，当居民更改姓名、地址、电话号码等信息时，相应的项目也要被修改。图1.1 展示了一个简单的数据库。

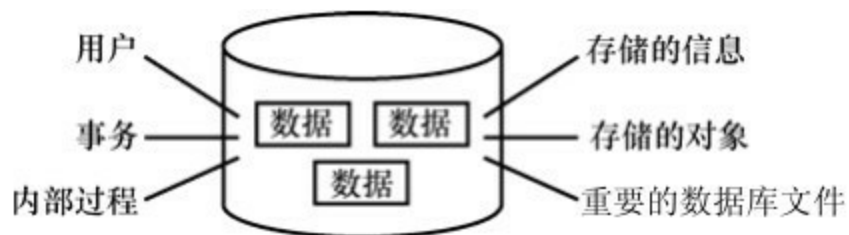


图1.1 数据库

### 1.1.5 关系型数据库

关系型数据库由被称为表的逻辑单元组成，这些表在数据库内部彼此关联。关系型数据库可以将数据分解为较小的、可管理的逻辑单元，从而在公司这一级别上更易维护，并提供更优化的数据库性能。如图1.2所示，表之间通过共同的关键字（数据值）彼此关联。

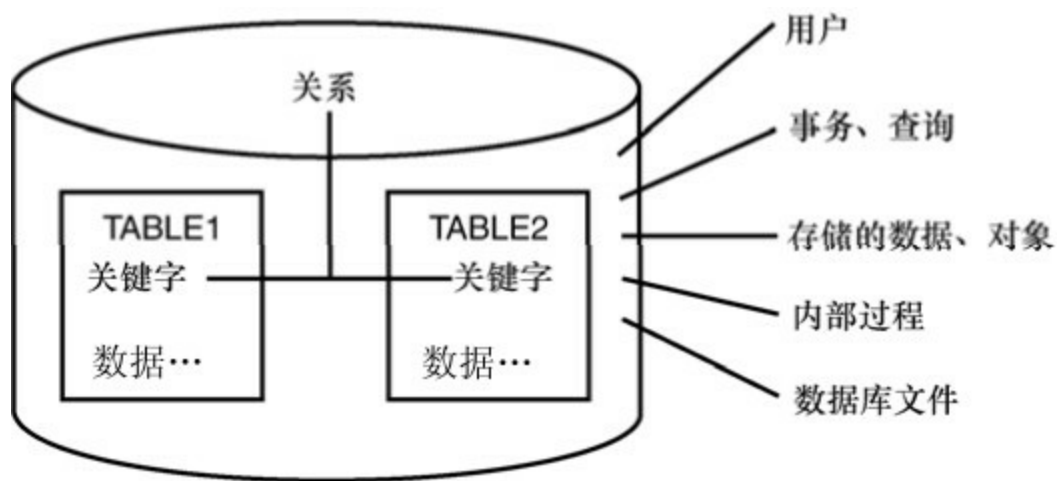


图1.2 关系型数据库

由于关系型数据库里的表是相互关联的，所以通过一个查询可以获得足够的数据库数据（虽然需要的数据可能处于多个表里）。由于关系型数据库的表之间可以具有共同的关键字或字段，所以多个表里的数据可以结合在一起形成一个数据集。本书后续内容会不断展示关系型数据库的优越之处，包括整体性能和方便的数据访问。

### 1.1.6 客户端/服务器技术

过去，计算机业由大型机统治着，它们是体积庞大、功能强悍的系统，具有大容量存储和高速数据处理能力。用户通过哑终端与主机通信，所谓哑终端就是没有处理能力的终端，依靠主机的CPU、外设和内存进行工作。每个终端通过一个数据链连接到主机。主机模式能够很好地实现其设计目的，并且在当今很多领域还在发挥作用，但另一种更伟大的技术出现了：客户端/服务器模型。

在客户端/服务器系统里，主机被称为服务器，可以通过网络进行访问（通常是局域网或广域网）。访问服务器的通常是个人计算机（PC）或其他服务器，而不是哑终端。每台个人计算机被称为客户端，通过网络与服务器进行通信。这也就是“客户端/服务器”名称的由来。客户端/服务器模型与主机模型之间最大的差别在于作为客户端的个人计算机能够自己“思考”，能够利用自身的CPU和内存处理数据，并且能够轻松地通过网络访问服务器。在大多数情况下，客户端/服务器模型可以适用于当今全部商业需求。

现代数据库系统位于多种不同的操作系统之上，而这些操作系统又运行在多种不同的计算机上。最常见的操作系统有基于Windows的系统、Linux和像UNIX这样的命令行系统。数据库主要位于客户端/服务器和Web环境里。不能实现数据库系统的主要原因是缺乏培训和经验。对客户端/服务器模型和基于Web的系统的理解已经与互联网技术开发和网络计算一起成为当今商业的强制要求（虽然有时显得不合理）。图1.3展示了客户端/服务器技术的概念。



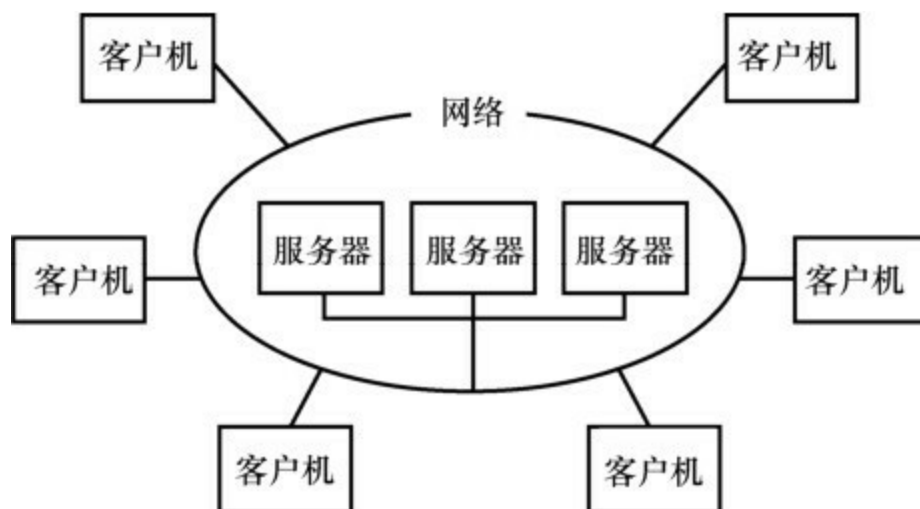


图1.3 客户端/服务器模型

### 1.1.7 基于Web的数据库系统

商业信息系统正在向Web迁移。现在我们能够通过互联网访问数据库，这意味着使用浏览器（比如 IE 和 Firefox）就能访问公司的信息。顾客（数据的用户）能够定购货物、查看存货、查看订单状态、修改账目、转账等。

顾客只需打开浏览器，访问公司的站点、登录，就可以利用公司页面内置的程序访问数据。大多数公司要求用户注册，并且为顾客提供登录名和密码。

当然，通过浏览器访问数据库的幕后工作并不像看上去这么简单。举例来说，Web程序可以运行SQL，从而访问公司的数据库，向Web服务器返回数据，然后再将数据返回到顾客的浏览器。

从用户的角度来说，基于 Web 的数据库系统类似于客户端/服务器系统。每个用户拥有一台客户机，安装了浏览器程序，能够连接到互联网。图1.3 所示的网络是互联网，这并不是必需的。在大多数情况下，客户机访问服务器是为了获取信息，并不关心服务器是否位于另一个州，甚至是另一个国家。基于Web的数据库系统的主要目的在于利用似

乎没有物理界限的数据库系统，提高数据可访问性，扩大公司的客户群。

### 1.1.8 主流数据库厂商

当今主流数据库厂商包括Oracle、Microsoft、Informix、Sybase和IBM。这些厂商以昂贵的基本许可费用出售各种版本的关系型数据库的闭源版本。其他一些厂商提供SQL数据库（关系型数据库）的开源版本，这些厂商包括MySQL、PostgreSQL和SAP。虽然还有其他很多厂商，但在此列出的这些名称经常会出现于图书、报纸、杂志、股市和互联网上。

每个厂商的SQL实现都是与众不同、独一无二的。数据库服务器就是一个产品——像市场上的其他产品一样，由多个不同的厂商生产。为了实现可移植性和易用性，厂商都保证其实实现兼容于当前的ANSI标准。如果一家公司从一个数据库服务器迁移到另一个时需要用户学习另一种语言来保持数据库功能，那就太糟糕了。

但是，每个厂商的SQL实现都针对其数据库服务器进行了增强，这些增强，或称之为扩展，是一些额外的命令和选项，附加于标准SQL软件包上，由特定的实现提供。

## 1.2 SQL会话

SQL会话是用户利用SQL命令与关系型数据库进行交互时发生的事情。当用户与数据库创建连接时，会话就被创建了。在SQL会话范围之内，用户可以输入有效的SQL命令对数据库进行查询，操作数据库里的数据，定义数据库结构（比如表）。会话可以通过直接与数据库创建连接来申请，也可以通过前端程序来申请。无论何种情况，会话通常是由通过网络访问数据库的用户在终端或工作站创建的。



### 1.2.1 CONNECT

当用户连接到数据库时，SQL会话就被初始化了。命令CONNECT用于创建与数据库的连接，它可以申请连接，也可以修改连接。举例来说，如果目前以 user1 的身份连接到数据库，我们还可以用CONNECT命令以user2的身份连接到数据库；连接成功之后，用于user1的SQL会话就被隐含地断开了。连接数据库通常需要用到以下命令：

```
CONNECT user@database
```

在尝试连接到数据库时，用户会看到一个提示，要求输入与当前用户名对应的密码。用户名用于向数据库说明身份，而密码是允许进行访问的钥匙。

### 1.2.2 DISCONNECT和EXIT

当用户与数据库断开连接时，SQL会话就被结束了。命令DISCONNECT用于断开用户与数据库的连接。当中断与数据库的连接之后，用户所使用的程序可能显得还在与数据库通信，但实际上已经没有连接了。当使用EXIT命令离开数据库时，SQL会话就结束了，而且用于访问数据库的软件通常会关闭。

```
DISCONNECT
```

## 1.3 SQL命令的类型

下面将讨论执行各种功能的SQL命令的基本分类。这些功能包括绑定数据库对象、操作对象、用数据填充数据库表、更新表里的现有数据、删除数据、执行数据库查询、控制数据库访问和数据库管理。

主要的分类包括：

数据定义语言（DDL）；

数据操作语言（DML）；  
数据查询语言（DQL）；  
数据控制语言（DCL）；  
数据管理命令；  
事务控制命令。

### [1.3.1 定义数据库结构](#)

数据定义语言（DDL）用于创建和重构数据库对象，比如创建和删除表。

本书要讨论的一些最基础的DDL命令包括：

- ▶ CREATE TABLE
- ▶ ALTER TABLE
- ▶ DROP TABLE
- ▶ CREATE INDEX
- ▶ ALTER INDEX
- ▶ DROP INDEX
- ▶ CREATE VIEW
- ▶ DROP VIEW

这些命令将在第3章、第17章和第20章中详细讨论。

### [1.3.2 操作数据](#)

数据操作语言（DML）用于操作关系型数据库对象内部的数据。  
3个基本DML命令是：

- ▶ INSERT
- ▶ UPDATE
- ▶ DELETE

这些命令将在第5章中详细讨论。

### [1.3.3 选择数据](#)

虽然只具有一个命令，但数据查询语言（DQL）是现代关系型数据库用户最关注的部分，它的基本命令是SELECT。

这个命令具有很多选项和子句，用于构成对关系型数据库的查询。查询是对数据库进行的信息调查，一般通过程序界面或命令行提示符向数据库发出。无论是简单的还是复杂的查询，含糊的还是明确的查询，都可以轻松地实现。

这个命令将在第7章到第16章中充分介绍。

### [1.3.4 数据控制语言](#)

SQL里的数据控制语言用于控制对数据库里数据的访问。这些数据控制语言（DCL）命令通常用于创建与用户访问相关的对象，以及控制用户的权限。这些控制命令包括：

- ▶ ALTER PASSWORD
- ▶ GRANT
- ▶ REVOKE
- ▶ CREATE SYNONYM

这些命令通常与其他命令组合在一起，在本书多个章节中都有介绍。

### [1.3.5 数据管理命令](#)

数据管理命令用于对数据库里的操作进行审计和分析，还有助于分析系统性能。常用的两个数据管理命令如下所示：

► START AUDIT

► STOP AUDIT

不要把数据管理与数据库管理混为一谈。数据库管理是对数据库的整体管理，它包括各级命令的使用。对于不同的SQL实现来说，数据管理与SQL语言的核心命令相比具有更明显的独特性。

### [1.3.6 事务控制命令](#)

除了前面介绍的几类命令，下面这些命令可以用于管理数据库事务。

**COMMIT：**保存数据库事务。

**ROLLBACK：**撤销数据库事务。

**SAVEPOINT：**在一组事务里创建标记点以用于回退（ROLLBACK）。

**SET TRANSACTION：**设置事务的名称。

事务命令将在第6章中详细讨论。

## [1.4 本书使用的数据库](#)

在继续讨论SQL基础知识之前，我们先来介绍一下本书后续课程中要使用的数据库。下面的小节会介绍所用的表，说明它们之间的关系、它们的结构，并展示其中包含的数据。

图1.4展示了本书范例、测验和练习中所用的表的关系。每个表都有不同的名称、包含一些字段。图中的映射线表示了特定表之间通过共

享字段（通常被称为主键）创建的联系。

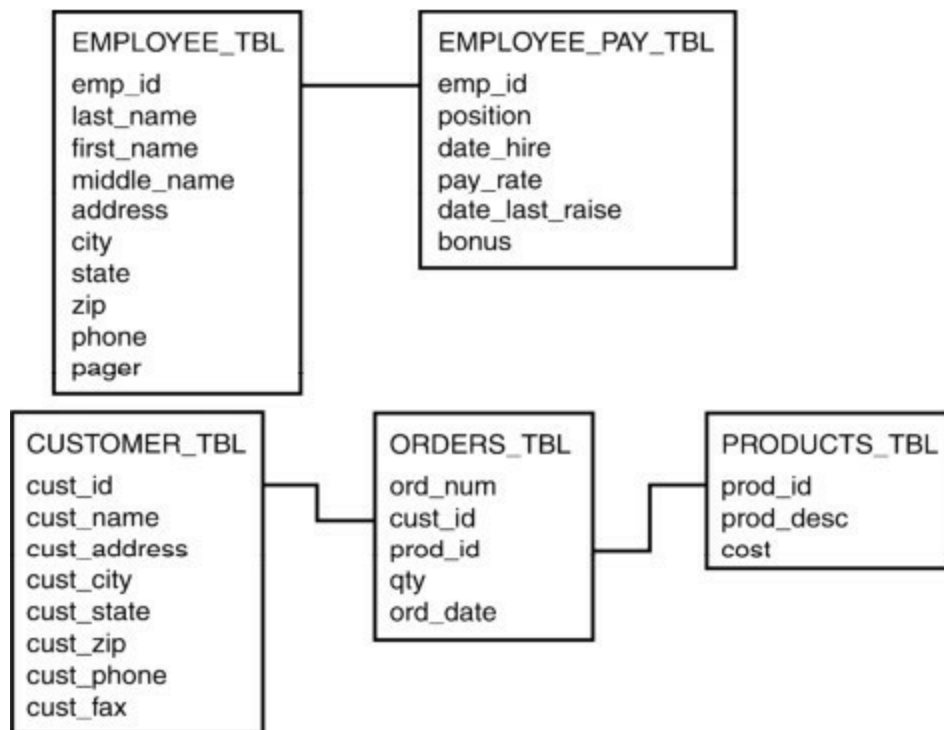


图1.4 本书所用表之间的关系

### 1.4.1 表命名标准

像商业活动中的其他标准一样，表命名标准对于保持良好的控制也是非常重要的。从前面对于表和数据的介绍可以看出，每个表的名称都以\_TBL 作为后缀，这种方式也是很多站点所采用的。后缀\_TBL 说明这个对象是个表，而关系型数据库里存在着多种不同类型的对象。例如，在后续章节会出现后缀\_INX，这说明对象是表的索引。命名标准几乎存在于整个机制之内，对任何关系型数据库的管理都起到了重要的辅助作用。需要说明的是，在命名数据库对象时，并不是一定要使用后缀。所谓的命名标准，只是为了在创建对象的时候有一定的准则可以用来遵循。读者可以根据自己的喜好来自由选择命名标准。

注意：不仅要遵循 SQL 实现的对象命名规则，还要符合本地商业

规则，从而创建出具有描述性的、与业务数据相关联的名称。

### 1.4.2 数据一瞥

下面将展示本书所用表里包含的数据。请花一些时间来研究这些数据，观察它们的区别，了解数据之间和表之间的关系。其中有些字段不是一定要包含数据，这是在创建表时指明的。

#### EMPLOYEE\_TBL

EMP_ID	LAST_NAME	FIRST_NAME	M	ADDRESS	CITY	ST	ZIP	PHONE
311549902	STEPHENS	TINA		D RR 3 BOX 17A	GREENWOOD	IN	47890	3178784465
442346889	PLEW	LINDA		C 3301 BEACON	INDIANAPOLIS	IN	46224	3172978990
213764555	GLASS	BRANDON	S	1710 MAIN ST	WHITELAND	IN	47885	3178984321
313782439	GLASS	JACOB		3789 RIVER BLVD	INDIANAPOLIS	IN	45734	3175457676
220984332	WALLACE	MARIAH		7889 KEYSTONE	INDIANAPOLIS	IN	46741	3173325986
443679012	SPURGEON	TIFFANY		5 GEORGE COURT	INDIANAPOLIS	IN	46234	3175679007

#### EMPLOYEE\_PAY\_TBL

EMP_ID	POSITION	DATE_HIRE	PAY_RATE	DATE_LAST	SALARY	BONUS
311549902	MARKETING	23-MAY-1999		01-MAY-2009	4000	
442346889	TEAM LEADER	17-JUN-2000	14.75	01-JUN-2009		
213764555	SALES MANAGER	14-AUG-2004		01-AUG-2009	3000	2000
313782439	SALESMAN	28-JUN-2007			2000	1000

220984332 SHIPPER	22-JUL-2006	11 01-JUL-2009
443679012 SHIPPER	14-JAN-2001	15 01-JAN-2009

CUSTOMER\_TBL

CUST_ID	CUST_NAME	ADDRESS	CUST_CITY	ST	ZIP	CUST_PHONE
CUST_FAX						
232	LESLIE GLEASON	798 HARDAWAY DR	INDIANAPOLIS	IN	47856	3175457690
109	NANCY BUNKER	APT A 4556 WATERWAY BROAD	RIPPLE	IN	47950	3174262323
345	ANGELA DOBKO	RR3 BOX 76	LEBANON	IN	49967	7658970090
090	WENDY WOLF	3345 GATEWAY DR	INDIANAPOLIS	IN	46224	3172913421
12	MARYS GIFT SHOP	435 MAIN ST	DANVILLE	IL	47978	3178567221
3178523434						
432	SCOTTYS MARKET	RR2 BOX 173	BROWNSBURG	IN	45687	3178529835
3178529836						
333	JASONS AND DALLAS GOODIES	LAFAYETTE SQ MALL	INDIANAPOLIS	IN	46222	3172978886 3172978887
21	MORGANS CANDIES AND TREATS	5657 W TENTH ST	INDIANAPOLIS	IN	46234	3172714398
43	SCHYLERS NOVELTIES	17 MAPLE ST	LEBANON	IN	48990	3174346758
287	GAVINS PLACE	9800 ROCKVILLE RD	INDIANAPOLIS	IN	46244	3172719991
3172719992						
288	HOLLYS GAMEARAMA	567 US 31	WHITELAND	IN	49980	3178879023
590	HEATHERS FEATHERS AND THINGS	4090 N SHADELAND AVE	INDIANAPOLIS	IN	43278	3175456768
610	REGANS HOBBIES	451 GREEN	PLAINFIELD	IN	46818	3178393441
3178399090						
560	ANDYS CANDIES	RR 1 BOX 34	NASHVILLE	IN	48756	8123239871
221	RYANS STUFF	2337 S SHELBY ST	INDIANAPOLIS	IN	47834	3175634402
175	CAMERON'S PIES	178 N TIBBS	AVON	IN	46234	3174543390
290	CALEIGH'S KITTENS	244 WEST ST	LEBANON	IN	47890	3174867754
56	DANIELS SPANIELS	17 MAIN ST	GREENWOOD	IN	46578	3172319908
978	AUTUMN'S BASKETS	5648 CENTER ST	SOUTHPORT	IN	45631	3178887565

ORDERS\_TBL

ORD_NUM	CUST_ID	PROD_ID	QTY	ORD_DATE
56A901	232	11235	1	22-OCT-2009
56A917	12	907	100	30-SEP-2009
32A132	43	222	25	10-OCT-2009



16C17	090	222	2	17-OCT-2009
18D778	287	90	10	17-OCT-2009
23E934	432	13	20	15-OCT-2009

PRODUCTS\_TBL

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.10
90	LIGHTED LANTERNS	14.50
15	ASSORTED COSTUMES	10.00
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95

### 1.4.3 表的构成

存储和维护有价值的数据是数据库存在的原因。前面的数据是用来解释本书中的SQL概念的，下面进一步详细介绍表里的元素。记住，表是数据存储的最常见和最简单的形式。

#### 一、字段

每个表都可以分解为更小的项，这些项被称为“字段”。字段是表里的一列，用于保持每条记录的特定信息。表PRODUCTS\_TBL里的字段包括PROD\_ID、PROD\_DESC和COST。这些字段对表中的信息进行分类保存。

#### 二、记录或一行数据

记录，也被称为一行数据，是表里的各行。以表PRODUCTS\_TBL为例，它的第一行记录如下所示：

11235	WITCH COSTUME	29.99
-------	---------------	-------

很明显，这条记录由产品标识、产品描述和单价组成，对于每一种



不同的产品，表PRODUCTS\_TBL里都有一条相应的记录。

在关系型数据库的表里，一行数据是指一条完整的记录。

### 三、列

列是表里垂直的一项，包含表里特定字段的全部信息。举例来说，表 PRODUCTS\_TBL里代表产品描述的一列包含以下内容：

```
WITCH COSTUME  
PLASTIC PUMPKIN 18 INCH  
FALSE PARAFFIN TEETH  
LIGHTED LANTERNS  
ASSORTED COSTUMES  
CANDY CORN  
PUMPKIN CANDY  
PLASTIC SPIDERS  
ASSORTED MASKS
```

这一列基于字段PROC\_DESC，也就是产品描述。一列包含了表里每条记录中特定字段的全部信息。

### 四、主键

主键用于区分表里每一条数据行。表 PRODUCTS\_TBL 里的主键是 PROD\_ID，它通常是在表创建过程中初始化的。主键的特性确保了所有产品标识都是唯一的，也就是说表PRODUCTS\_TBL里每条记录都具有不同的PROD\_ID。主键避免了表中有重复的数据，并且还具有其他用途，具体介绍请见第3章。

### 五、NULL值

NULL是表示“没有值”的专用术语。如果表中某个字段的值是NULL，其表现形式就是字段为空，其值就是没有值。NULL并不等同于0或空格。值为NULL的字段在表创建过程中会保持为空。比如在表EMPLOYEE\_TBL里，并不是每个雇员的姓名里都有中间名，其相应字段的值就是NULL。

后续两章将详细介绍表里的其他元素。

#### [1.4.4 范例和练习](#)

本书中的很多练习使用MySQL、Microsoft SQL Server和Oracle数据库来生成范例。这三种数据库都具有免费版本，可以自由选择一种来安装，以便完成本书所设置的练习。但是由于这三种数据库都不能与SQL-2008完全兼容，因此练习的结果可能会有一些细小的差别，不完全复合ANSI标准。不过，掌握了基本的ANSI标准以后，读者就可以在不同的数据库实现之间进行自由切换，以便解决大部分的问题了。

### [1.5 小结](#)

前面介绍了SQL标准语言，简要说明了其历史，粗略展示了这个标准在过去是如何进化的。另外还讨论了数据库系统和当今技术，包括关系型数据库、客户端/服务器系统、基于Web的系统，这些对于理解SQL都是非常重要的。还介绍了SQL语言的主要组件，说明了关系型数据库市场里有众多的厂商，当然也就有多种各具特色的SQL实现。虽然它们与ANSI SQL都略有不同，但大多数厂商都在一定范围内遵循当前标准（SQL-2008），后者维护了SQL的一致性，让SQL程序具有可移植性。

另外还介绍了本书所使用的数据库。从前面的内容可以看到，数据库由一些表组成，它们彼此有一定的关联；我们也看到此时表中包含的数据。本章还介绍了SQL的一些背景知识，展示了现代数据库的概念。在完成本章的练习之后，读者会信心十足地继续后面的课程。

### [1.6 问与答](#)

问：如果要学习SQL，是不是可以使用SQL的任何一种实现呢？

答：是的，只要数据库的实现是兼容ANSI SQL的，我们就可以与

之交互。如果实现并不是完全兼容的，我们只需要稍作调整即可。

问：在客户端/服务器环境里，个人计算机是客户端，还是服务器？

答：个人计算机被认为是客户端，但有时服务器也可以充当客户端的角色。

问：创建每一个表都必须使用**\_TBL**作为名称后缀吗？

答：当然不是。使用**\_TBL**作为表名称后缀是我们所选择的一种命名方式，能够方便地标识出数据库里的表。当然还可以把**TBL**完整拼写为**TABLE**，或是避免使用后缀，比如**EMPLOYEE\_TBL**可以只命名为**EMPLOYEE**。

## 1.7 实践

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习是为了把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### 1.7.1 测验

1. 缩写“SQL”的含义是什么？
2. SQL命令的6个主要类别是什么？
3. 4个事务控制命令是什么？
4. 对于数据库访问来说，客户端/服务器模型与Web技术之间的主要区别是什么？
5. 如果一个字段被定义为NULL，这是否表示这个字段必须要输入某些内容？

### 1.7.2 练习

1. 说明下面的SQL命令分别属于哪个类别：

```
CREATE TABLE
DELETE
SELECT
INSERT
ALTER TABLE
UPDATE
```

2. 观察下面这个表，选出适合作为主键的列：

<b>EMPLOYEE_TBL</b>	<b>INVENTORY_TBL</b>	<b>EQUIPMENT_TBL</b>
name	item	model
phone	description	year
start date	quantity	serial number
address	item number	equipment number
employee number	location	assigned to

3. 参考附录B，选择一种数据库实现，下载并安装好，为后面的练习做准备。

## [第二部分 创建数据库](#)

第2章 定义数据结构

第3章 管理数据库对象

第4章 规格化过程

第5章 操作数据

第6章 管理数据库事务

### [第2章 定义数据结构](#)

本章的重点包括：

概述表的底层数据

简介基本的数据类型

使用不同类型的数据

展示不同数据类型之间的区别

在本章中，我们将进一步研究前一章结尾时所展示的数据，讨论数据本身的特征及其如何保存在关系型数据库里。数据类型有多种，稍后就会介绍。

#### [2.1 数据是什么](#)

数据是一个信息集合，以某种数据类型保存在数据库里。数据包括姓名、数字、货币、文本、图形、小数、计算、统计等，几乎涵盖能够想象到的任何东西。数据可以保存为大写、小写或大小写混合，数据可以被操作或修改，大多数数据在其生存周期内不会保持不变。

数据类型用于指定特定列所包含数据的规则，它决定了数据保存在列里的方式，包括分配给列的宽度，以及值是否可以是字母、数字、日

期和时间等。任何数据或数据的组合都有对应的数据类型，这些数据类型用于存储像字母、数字、日期和时间、图像、二进制数据等。更详细地说，数据可以包括姓名、描述、数字、计算、图像、图像描述、文档等。

数据是数据库的意义所在，必须受到保护。数据的保护者就是数据库管理员（DBA），但每个数据库用户也有责任采取必要手段来保护数据。关于数据安全的内容将在第 18 章和第19章详细讨论。

## 2.2 基本数据类型

本节将介绍ANSI SQL支持的基本数据类型。数据类型是数据本身的特征，其特性被设置到表里的字段。举例来说，我们可以指定某个字段必须包含数字值，不允许输入由数字或字母组成的字符串；我们也不希望在保存货币数值的字段里输入字母。为数据库里每个字段定义数据类型可以大幅减少数据库里由于输入错误而产生的错误数据。字段定义（数据类型定义）是一种数据检验方式，控制了每个字段里可以输入的数据。

在一些RDBMS实现里，一些数据类型可以根据其格式自动转化为其他数据类型，这种转换被称为隐式转换，表示数据库会自动完成转换。举例来说，从一个数值字段取出一个数值1000.92，把它输入到一个字符串字段里，此时数据库就会完成自动转换。其他一些数据类型不能由主机 RDBMS 隐式转换，就必须经过显式转换，这通常需要调用 SQL 函数，比如CAST或CONVERT，如下所示：

```
SELECT CAST('12/27/1974' AS DATETIME) AS MYDATE
```

像其他大多数语言一样，最基本的数据类型是：  
字符串类型；

数值类型；

日期和时间类型。

提示：**SQL**数据类型

SQL 的每个实现都具有自己的数据类型集。使用某个实现所特有的数据类型是必要的，以支持每个实现处理存储数据的方式策略。但基本数据类型在不同实现之间还是相同的。

### 2.2.1 定长字符串

定长字符串通常具有相同的长度，是使用定长数据类型保存的。下面是 SQL 定长字符串的标准：

**CHARACTER(*n*)**

*n*是一个数字，定义了字段里能够保存的最多字符数量。

有些SQL实现使用CHAR数据类型来保存定长数据。字母可以保存到这种数据类型里。州名缩写就是定长数据类型的一个例子，因为所有的缩写都由两个字母组成。

在定长数据类型里，通常使用空格来填充数量不足的字符。举例来说，如果字段长度是10，而输入的数据只有5位，那么剩余5位就会被记录为空格。填充空格确保了字段里每个值都具有相同的长度。

警告：定长数据类型

不要使用定长数据类型来保存长度不定的数据，比如姓名。如果不恰当地使用定长数据类型，可能会导致浪费可用空间，以及影响对不同的数据进行精确比较。

应该使用变长数据类型来保存长度不定的字符串，从而节省数据库空间。

### 2.2.2 变长字符串

SQL支持变长字符串，也就是长度不固定的字符串。下面是SQL变长字符串的标准：

**CHARACTER VARYING(*n*)**

*n*是一个数字，表示字段里能够保存的最多字符数量。

常见的变长字符串数据类型有VARCHAR、VARINARY和VARCHAR2。VARCHAR是ANSI标准，Microsoft SQL Server和MySQL也使用它；VARINARY和VARCHAR2都是由Oracle使用的。定义为字符的字段里可以保存数字和字母，这意味着数据中可能包含数字字符。VARBINARY类似于VARCHAR和VARCHAR2，只是它包含的是长度不定的字节。这种数据类型通常被用来保存数字式数据，例如图像文件。

定长数据类型利用空格来填充字段里的空白，但变长字符串不这样做。举例来说，如果某个变长字段的长度定义为10，而输入的字符串长度为5，那么这个值的总长度也就是5，这时并不会使用空格来填充字段里的空白。

### 2.2.3 大对象类型

有些变长数据类型需要保存更长的数据，超过了一般情况下为VARCHAR字段所保留的长度，比如现在常见的BLOB和TEXT数据类型。这些数据类型是专门用于保存大数据集的。BLOB是二进制大对象，它的数据是很长的二进制字符串（字节串）。BLOB适合在数据库里存储二进制媒体文件，比如图像和MP3。

TEXT数据类型是一种长字符串类型，可以被看作一个大VARCHAR字段，通常用于在数据库里保存大字符集，比如博客站点的HTML输入。在数据库里保存这种类型的数据可以实现站点的动态更新。



### 2.2.4 数值类型

数值被保存在定义为某种数值类型的字段里，一般包括NUMBER、INTEGER、REAL、DECIMAL等。

下面是SQL数值的标准：

- ▶ `BIT(n)`
- ▶ `BIT VARYING(n)`
- ▶ `DECIMAL(p,s)`
- ▶ `INTEGER`
- ▶ `SMALLINT`
- ▶ `BIGINT`
- ▶ `FLOAT(p,s)`
- ▶ `DOUBLE PRECISION(p,s)`
- ▶ `REAL(s)`

*p*表示字段的最大长度。

*s*表示小数点后面的位数。

SQL实现中一个通用的数值类型是NUMERIC，它符合ANSI标准。数值可以是0、正值、负值、定点数和浮点数。下面是使用NUMERIC的一个范例：

`NUMERIC(5)`

这个命令把字段能够接受的最大值限制为 99 999。在本书范例所涉及的数据库实现中，NUMERIC都是以DECIMAL类型实现的。

### 2.2.5 小数类型

小数类型是指包含小数点的数值。SQL 的小数标准如下所示，其中 p 表示有效位数，s表示标度。

**DECIMAL (p,s)**

有效位数是数值的总体长度。举例来说，在数值定义 DECIMAL(4,2)里，有效位数是4，也就是说数值总位数是4。标度是小数点后面的位数，在前例中是2。如果实际数值的小数码数超出了定义的位数，数值就会被四舍五入。比如34.33写入到定义为DECIMAL(3,1)的字段时，会被四舍五入为34.3。

如果数值按照如下方式被定义，其最大值就是99.99：

**DECIMAL (4,2)**

有效位数是4，表示数值的总体长度是4；标度是2，表示小数点后面保留2位。小数点本身并不算作一个字符。

定义为DECIMAL(4,2)的字段允许输入的数值包括：

- ▶ 12
- ▶ 12.4
- ▶ 12.44
- ▶ 12.449

最后一个值 12.449 在保存到字段时会被四舍五入为 12.45。在这种定义下，任何12.445~12.449之间的数值都会被四舍五入为12.45。

### 2.2.6 整数

整数是不包含小数点的数值（包括正数和负数）。

下面是一些有效的整数：

- ▶ 1
- ▶ 0
- ▶ -1
- ▶ 99
- ▶ -99
- ▶ 199

### [2.2.7 浮点数](#)

浮点数是有效位数和标度都可变并且没有限制的小数数值，任何有效位数和标度都是可以的。数据类型REAL代表单精度浮点数值，而DOUBLE PRECISION表示双精度浮点数值。单精度浮点数值的有效位数为1~21（包含），双精度浮点数值的有效位数为22~53（包含）。下面是一些FLOAT数据类型的范例：

- ▶ FLOAT
- ▶ FLOAT(15)
- ▶ FLOAT(50)

### [2.2.8 日期和时间类型](#)

日期和时间数据类型很显然是用于保存日期和时间信息的。标准SQL支持DATETIME数据类型，它包含以下类型：

- ▶ DATE
- ▶ TIME
- ▶ DATETIME
- ▶ TIMESTAMP

DATETIME数据类型的元素包括：

- ▶ YEAR
- ▶ MONTH
- ▶ DAY
- ▶ HOUR
- ▶ MINUTE
- ▶ SECOND

注意：日期和时间类型

SECOND元素还可以再分解为几分之一秒，其范围是00.000~61.999，但并不是所有SQL实现都支持这个范围。多出来的1.999秒是用于实现闰秒的。

每种SQL实现可能都具有自定义的数据类型来保存日期和时间。前面介绍的数据类型和元素是每个SQL厂商都应该遵守的标准，但大多数实现都具有自己的数据类型来保存日期值，其形式与实际存储方式有所不同。

日期数据一般不指定长度。稍后我们会更详细地介绍日期类型，包括日期信息在某些实现里的保存方式、如何使用转换函数操作日期和时间，并且用范例展示在实际工作中如何使用日期和时间。

### [2.2.9 直义字符串](#)

直义字符串就是一系列字符，比如姓名或电话号码，这是由用户或程序明确指定的。直义字符串包含的数据与前面介绍的数据类型具有一样的属性，但字符串的值是已知的。列本身的价值通常是不能确定的，因为每一列通常包含了字段在全部记录里的不同值。

实际上并不需要把字段指定为直义字符串数据类型，而是指定字符

串。直义字符串的范例如下所示：

```
▶ 'Hello'
▶ 45000
▶ "45000"
▶ 3.14
▶ 'November 1, 1997'
```

字符型的字符串由单引号包围，数值45000没有用单引号包围，而第二个45000用双引号包围了。一般来说，字符型字符串需要使用单引号，而数值型不需要。

将一个数据转换成数值类型的过程属于隐式转换。在这个过程中，数据库会自动判断应该使用哪种数据类型。所以，如果一个数据没有使用单引号包围起来，那么SQL程序就会将其认定为数值类型。因此，必须要特别留意数据的形式。否则，存储结果可能出现偏差，或者报错。稍后将介绍如何在数据库查询里使用直义字符串。

### 2.2.10 NULL数据类型

第1章已经介绍过，NULL值表示没有值。NULL值在SQL里有广泛的应用，包括表的创建、查询的搜索条件，甚至是在直义字符串里。

下面是两种引用NULL值的方法：

NULL（关键字NULL本身）；

下面这种形式并不代表NULL值，它只是一个包含字符N-U-L-L的直义字符串：

```
'NULL'
```

在使用NULL数据类型时，需要明确它表示相应字段不是必须要输

入数据的。如果某个字段必须包含数据，就把它设置为NOT NULL。只要字段有可能不包含数据，最好就把它设置为NULL。

### [2.2.11 布尔值](#)

布尔值的取值范围是TRUE、FALSE和NULL，用于进行数据比较。举例来说，在查询中设置条件时，每个条件都会被求值，得到TRUE、FALSE或NULL。如果查询中所有条件的值都是TRUE，数据就会被返回；如果某个条件的值是FALSE或NULL，数据就不会返回。

比如下面这个范例：

```
WHERE NAME = 'SMITH'
```

这可能是查询里的一个条件，目标表里每行数据都根据这个条件进行求值。如果表里某行的NAME字段值是SMITH，条件的值就是TRUE，相应的记录就会被返回。

大多数数据库实现并没有一个严格意义上的BOOLEAN类型，而是代之以各自不同的实现方法。MySQL拥有BOOLEAN类型，但实质上与其现有的TINYINT类型相同。Oracle倾向于让用户使用一个CHAR(1)值来代替布尔值，而SQL Server则使用BIT来代替。

注意：数据类型实现上的差异

前面介绍的这些数据类型在不同的SQL实现里可能具有不同的名称，但其概念是通用的。其中大多数数据类型得到了大多数关系型数据库的支持。

### [2.2.12 自定义类型](#)

自定义类型是由用户定义的类型，它允许用户根据已有的数据类型来定制自己的数据类型，从而满足数据存储的需要。自定义类型极大地丰富了数据存储的可能性，使开发人员在数据库程序开发过程中具有更

大的灵活性。语句CREATE TYPE用于创建自定义类型。

举例来说，在MySQL和Oracle中，可以像下面这样创建一个类型：

```
CREATE TYPE PERSON AS OBJECT
(NAME      VARCHAR (30),
SSN        VARCHAR (9));
```

然后可以像下面这样引用自定义类型：

```
CREATE TABLE EMP_PAY
(EMPLOYEE   PERSON,
SALARY      DECIMAL(10,2),
HIRE_DATE   DATE);
```

表EMP\_PAY第一列EMPLOYEE的类型是PERSON，这正是在前面创建的自定义类型。

### [2.2.13 域](#)

域是能够被使用的有效数据类型的集合。域与数据相关联，从而只接受特定的数据。在域创建之后，我们可以向域添加约束。约束与数据类型共同发挥作用，从而进一步限制字段能够接受的数据。域的使用类似于自定义类型。

像下面这样就可以创建域：

```
CREATE DOMAIN MONEY_D AS NUMBER(8,2);
```

像下面这样为域添加约束：

```
ALTER DOMAIN MONEY_D
ADD CONSTRAINT MONEY_CON1
CHECK (VALUE > 5);
```

然后像下面这样引用域：

```
CREATE TABLE EMP_PAY
(EMP_ID      NUMBER(9),
EMP_NAME     VARCHAR2(30),
PAY_RATE     MONEY_D);
```

## [2.3 小结](#)

SQL具有多种数据类型，对于使用过其他编程语言的人来说，这些都不算陌生。数据类型允许不同类型的数据保存到数据库，比如单个字符、小数、日期和时间。无论是使用像 C 这样的第三代编程语言，还是使用关系型数据库实现 SQL 编码，数据类型的概念都是一样的。当然，不同实现中数据类型的名称可能有所不同，但其工作方式基本上是一样的。另外，关系型数据库管理系统并不是一定要实现 ANSI 标准里规定的全部数据类型才会被认为是与 ANSI 兼容的，因此最好查看具体实现的文档来了解可以使用的数据类型。

在考虑数据类型、长度、标度和精度时，一定要仔细地进行短期和长远的规划。另外，公司制度和希望用户以什么方式访问数据也是要考虑的因素。开发人员应该了解数据的本质，以及数据在数据库里是如何相互关联的，从而使用恰当的数据类型。

## [2.4 问与答](#)

问：当字段被定义为字符类型时，为什么还可以保存像个人社会保险号码这样的数字值呢？

答：字符串数据类型允许输入字母数字，而数字值当然是属于这个范围内的。这个过程被称为隐式转换，它是由数据库系统自动完成的。一般来说，只有用于计算的数字才以数值类型保存。但从另一方面来说，把全部数值字段都设置为数值类型有助于控制字段的输入数据。

问：定长和变长数据类型之间到底有什么区别呢？



答：假设我们把个人姓名里的姓字段定义为长度为20B的定义数据类型，而某人的姓是Smith。当这个数据进入表之后，会占据20B的空间，其中5B用于保存Smith，另外15B是额外的空格（因为这是定长数据类型）。如果使用长度为20B的变长数据类型，并且也输入Smith作为数据，那么它只会占据5B。想象一下，如果要添加100 000条记录，那么使用变长数据类型也许就会节省1.5MB。

问：数据类型的长度有限制吗？

答：当然有，而且不同实现中对此限制也是有所区别的。

## 2.5 实践

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### 2.5.1 测验

1. 判断对错：个人社会保险号码，输入格式为 '1111111111'，它可以是下面任何一种数据类型：定长字符、变长字符、数值。
2. 判断对错：数值类型的标度是指数值的总体长度。
3. 所有的SQL实现都使用同样的数据类型吗？
4. 下面定义的有效位数和标度分别是多少？

```
DECIMAL(4,2)  
DECIMAL(10,2)  
DECIMAL(14,1)
```

5. 下面哪个数值能够输入到定义为DECIMAL(4,1)的字段里？  
A. 16.2

- B. 116.2
- C. 16.21
- D. 1116.2
- E. 1116.21

6. 什么是数据？

### 2.5.2 练习

1. 考虑以下字段名称，为它们设置适当的数据类型，确定恰当的长度，并给出一些示范数据：

- a) ssn
- b) state
- c) city
- d) phone\_number
- e) zip
- f) last\_name
- g) first\_name
- h) middle\_name
- i) salary
- j) hourly\_pay\_rate
- k) date\_hired

2. 同样是这些字段，判断它们应该是NULL或NOT NULL。体会在不同的应用场合，有些一般是NOT NULL的字段可能应该是NULL，反之亦然。

- a) ssn
- b) state
- c) city
- d) phone\_number

- e) zip
- f) last\_name
- g) first\_name
- h) middle\_name
- i) salary
- j) hourly\_pay\_rate
- k) date\_hired

3. 现在要为后面的课程创建一个数据库。在此之前，先要安装一种数据库实现——MySQL、Oracle或者Microsoft SQL Server。

### MySQL

在Windows操作系统下，找到MySQL的安装目录，双击bin目录，再双击名为mysql.exe的可执行文件。如果看到错误消息说“server could not be found”，就首先从 bin 目录里执行winmysqladmin.exe，然后输入用户名和密码。在服务程序启动之后，再从bin目录里执行mysql.exe。

在mysql>提示符下，输入如下命令来创建本书练习所用的数据库：

```
create database learnsql;
```

命令输入后要按掉头键。在进行本书后续的练习时，我们先要运行mysql.exe，然后在命令提示符下输入如下命令来使用刚才创建的数据库：

```
use learnsql;
```

### Oracle

打开网页浏览器并进入管理主页，通常管理主页的网址是http://127.0.0.1:8080/apex。此时会出现登录界面，如果是第一次登录系统，用户名为system，密码为安装系统时，由用户所设置的密码。在管

理界面中，有SQL、SQL Commands和Enter Command三种运行方式可供选择。在命令行界面输入以下命令并按运行键：

```
create user learnsql identified by learnsql_2010;
```

在 Oracle 中创建一个用户后，系统会自动创建一个对应的规划（schema）。因此，运行上述命令后，在创建了一个用户的同时，也创建了一个名为learnsql的规划（schema）。Oracle中的规划（schema），相当于MySQL和Microsoft SQL Server中的数据库。退出系统以后，以新创建的用户身份重新登录系统，就可以看到规划（schema）。

#### Microsoft

在“开始”菜单的“运行”窗口中，输入 SSMS.exe 并按“确定”按钮。此时，开始运行SQL Server Management Studio。弹出的第一个对话框用于连接数据库。此处的服务器名称应该为“localhost”，如果系统没有自动显示出来，则可以手工输入。其余选项保持不变，单击“连接”按钮。此时，在界面的左侧会显示出用户的本地数据库实例。用鼠标右键单击“localhost”，在弹出的快捷菜单中选择“新建查询”选项后，界面右侧会打开一个查询窗口。输入以下命令并按F5键：

```
Create database learnsql;
```

用鼠标右键单击“localhost”下名为“数据库”的文件夹，在弹出的快捷菜单中选择“刷新”选项。之后单击文件夹前面的“+”符号展开文件夹，即可看到刚刚创建的名为“learnsql”的数据库。

### 第3章 管理数据库对象

本章重点包括：

数据库对象简介

规划简介

表简介

讨论表的实质与属性

创建和操作表的范例

讨论表存储选项

引用完整性和数据一致性的概念

本章将介绍数据库对象：它们是什么、它们的作用、它们如何存储、它们之间的关系。数据库对象是关系型数据库的底层构架，是数据库里保存信息的逻辑单元。本章介绍的内容主要是围绕表的，其他数据库对象将在后面的章节讨论。

### [3.1 什么是数据库对象](#)

数据库对象是数据库里定义的、用于存储或引用数据的对象，比如表、视图、簇、序列、索引和异名。本章的内容以表为主，因为它是关系型数据库里最主要、最简单的数据存储形式。

### [3.2 什么是规划](#)

规划是与数据库某个用户名相关联的数据库对象集合。相应的用户名被称为规划所有人，或是关联对象组的所有人。数据库里可以有一个或多个规划。用户只与同名规划相关联，通常情况下反之亦然。一般来说，当用户创建一个对象时，就是在自己的规划里创建了它，除非明确指定在另一个规划里创建它。因此，根据在数据库里的权限，用户可以创建、操作和删除对象。规划可以只包含一个表，也可以包含无数个对象，其上限由具体的SQL实现决定。

假设我们从管理员获得了一个数据库用户名和密码，用户名是

USER1。我们登录到数据库并创建一个名为 EMPLOYEE\_TBL 的表，这时对于数据库来说，表的实际名称是USER1.EMPLOYEE\_TBL，这个表的规划名是USER1，也就是这个表的所有者。这样，我们就为这个规划创建了第一个表。

当我们访问自己所拥有的表时（在自己的规划里），不必引用规划名称。举例来说，使用下面两种方式都可以引用刚才创建的表：

```
EMPLOYEE_TBL  
USER1.EMPLOYEE_TBL
```

我们当然喜欢使用第一种方法，因为它简单，需要敲击键盘的次数比较少。如果其他用户要访问这个表，就必须指定规划名称，如下所示：

```
USER1.EMPLOYEE_TBL
```

第 20 章将介绍如何分配权限，从而让其他用户访问我们的表。还会介绍异名的概念，也就是让表具有另一个名称，使我们在访问表时不必指定规划名。图3.1 展示了关系型数据库里的两个规划。

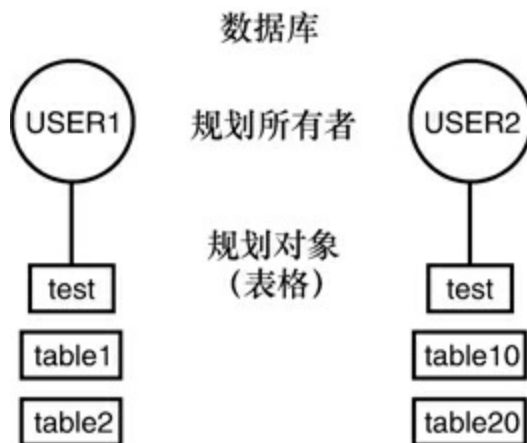


图3.1 数据库里的规划

图3.1里的数据库有两个用户账户：USER1和USER2。每个用户都有自己的规划，他们访问自己的表和对方的表的方式如下所示：

USER1访问自己的TABLE1： TABLE1

USER1访问自己的TEST： TEST

USER1访问USER2的TABLE10： USER2.TABLE10

USER1访问USER2的TEST： USER2.TEST

在这个范例里，两个用户都有一个名为TEST的表。在数据库里，不同的规划中可以具有名称相同的表。从另一个角度来说，表的名称里实际上包含着规划名，所以不同规划里表面上同名的表实际上具有不同的名称。比如USER1.TEST与USER2.TEST显然是不同的。如果在访问表时没有指定规划名，数据库服务程序会默认选择用户所拥有的表。也就是说，如果USER1要访问表TEST，数据库服务程序会先查找USER1拥有的名为TEST的表，然后再查找USER1拥有的其他对象，比如指向另一个规划里的表的异名。第21章会详细介绍异名的概念。用户必须明确理解自己规划内对象和规划外对象的区别，如果在执行修改表的操作时没有指定规划名，比如使用DROP命令，数据库会认为用户要操作自己规划里的表，这可能会导致意外删除错误的对象。因此，在进行数据库操作时，一定要注意自己是以什么身份登录到数据库的。

注意：对象命名规则在不同的数据库服务程序中有所差异

每个数据库服务程序都有命名对象和对象元素（比如字段）的规则，请查看具体实现的说明文档来了解详细要求。

### [3.3 表：数据的主要存储方式](#)

表是关系型数据库里最主要的数据存储对象，其最简单形式是由行和列组成，分别都包含着数据。表在数据库占据实际的物理空间，可以是永久的或是临时的。

### 3.3.1 列

字段在关系型数据库也被称为列，它是表的组成部分，被设置为特定的数据类型。数据类型决定了什么样的数据可以保存在相应的列中，从而确保了数据的完整性。

每个数据库表都至少要包含一列。列元素在表里用于保存特定类型的数据，比如人名或地址。举例来说，姓名就可以作为顾客表里一个有效的列。图3.2展示了表里的列。

EMP_ID	LAST_NAME	FIRST_NAME	MIDDLE_NAME
213764955	GLASS	BRANDON	SCOTT
220984332	WALLACE	MARIAH	DAVE
311549902	STEPHENS	TINA	DAWN
313782439	GLASS	JACOB	DAVE
442346889	PLEW	LINDA	CAROL
443679012	SPURGEON	TIFFANY	DAVE

图3.2 列的范例

一般来说，列的名称应该是连续的字符串，其长度在不同SQL实现中都有明确规定。我们一般使用下划线作为分隔符，比如表示顾客姓名的列可以命名为CUSTOMER\_NAME，它比CUSTOMERNAME更好一些。这样做可以提高数据库对象的可读性。读者也可以使用其他命名规则，例如驼峰匹配，以满足特定的需求。对于一个数据库开发团队来说，明确一个命名规则，并在开发的全过程中严格遵守这一规则，是非常重要的。

列中最常见的数据类型是字符串。这种数据可以保存为大写或小写字符，应该根据数据的使用方式具体选择。在大多数情况下，出于简化和一致的目的，数据是以大写存储的。如果数据库里存储的数据具有不同的大小写，我们可以根据需要利用函数把数据转化为大写或小写，具体函数将在第11章介绍。

列也可以指定为NULL或NOT NULL，当设置为NOT NULL时，表示其中必须包含数据；设置为NULL时，就表示可以不包含数据。



NULL不是空白，而是类似于一个空的字符串，在数据库中占据了一个特殊的位置。因此，如果某一个位置缺少数据，就可以使用NULL。

### [3.3.2 行](#)

行是数据库表里的一条记录。举例来说，顾客表里的一行数据可能包含顾客的标识号码、姓名、地址、电话号码、传真号码等。行由字段组成，表最少可以包含一行数据，也可以包含数以百万计的记录。图3.3展示了表里的行。

EMP_ID	LAST_NAME	FIRST_NAME	MIDDLE_NAME
213764555	GLASS	BRANDON	SCOTT
220384332	WALLACE	MARION	SCOTT
311549902	STEPHENS	TINA	DAWN
313782439	GLASS	JACOB	SCOTT
442346889	PLEW	LINDA	CAROL
443679012	SPURGEON	TIFFANY	SCOTT

图3.3 行的范例

### [3.3.3 CREATE TABLE语句](#)

SQL里的CREATE TABLE语句用于创建表。虽然创建表的实际操作十分简单，但在执行CREATE TABLE命令之前，应该花更多的时间和精力来设计表的结构，这样可以节省反复修改表结构而浪费的时间。

注意：本章所使用的数据类型

在本章的范例里，我们使用流行的数据类型CHAR（定长字符）、VARCHAR（变长字符）、NUMBER（数值，小数和整数）和DATE（日期和时间值）。

在创建表时，需要考虑以下一些基本问题。

表里会包含什么类型的数据？

表的名称是什么？

哪个（或哪些）列组成主键？

列（字段）的名称是什么？

每一列的数据类型是什么？

每一列的长度是多少？

表里哪些列可以是NULL？

注意：不同的系统往往有不同的命名规则

在命名对象和其他数据库元素时，一定要查看具体实现的规则。数据库管理员通常会采用某种“命名规范”来决定如何命名数据库里的对象，以便区分它们的用途。

在考虑了这些问题之后，实际的CREATE TABLE命令就很简单了。

创建表的基本语法如下所示：

```
CREATE TABLE table_name
( field1  data_type  [ not null ],
  field2  data_type  [ not null ],
  field3  data_type  [ not null ],
  field4  data_type  [ not null ],
  field5  data_type  [ not null ] );
```

在这个语句里，最后一个字符是分号。此外，括号是可选的。大多数SQL实现都以某些字符来结束命令，或是把命令发送到数据库服务程序。Oracle、Microsoft SQL Server和MySQL使用分号；而Transact-SQL、Microsoft SQL Server的ANSI SQL版本却不强制要求。不过，最好还是使用这样的字符来结束命令。在本书中，我们使用分号。

要创建一个名为EMPLOYEE\_TBL的表，使用MySQL语法规则的代码如下：

```

CREATE TABLE EMPLOYEE_TBL
(EMP_ID      CHAR(9)      NOT NULL,
EMP_NAME     VARCHAR (40) NOT NULL,
EMP_ST_ADDR  VARCHAR (20) NOT NULL,
EMP_CITY     VARCHAR (15) NOT NULL,
EMP_ST       CHAR(2)      NOT NULL,
EMP_ZIP      INTEGER(5)   NOT NULL,
EMP_PHONE    INTEGER(10)  NULL,
EMP_PAGER    INTEGER(10)  NULL);

```

下述代码同时适用于Oracle和Microsoft SQL Server:

```

CREATE TABLE EMPLOYEE_TBL
(EMP_ID      CHAR(9)      NOT NULL,
EMP_NAME     VARCHAR (40) NOT NULL,
EMP_ST_ADDR  VARCHAR (20) NOT NULL,
EMP_CITY     VARCHAR (15) NOT NULL,
EMP_ST       CHAR(2)      NOT NULL,
EMP_ZIP      INTEGER      NOT NULL,
EMP_PHONE    INTEGER      NULL,
EMP_PAGER    INTEGER      NULL);

```

这个表包含8列。列的名称中利用下划线对单词进行分隔（EMPLOYEE\_ID被缩写为EMP\_ID），这种方式可以让表和列的名称具有更好的易读性。每一个列都设置了数据类型和长度。同时，通过使用 NULL/NOT NULL，指定了哪些字段必须包含内容。EMP\_PHONE被定义为NULL，表示它的内容可以为空，因为有的人可能没有电话号码。各个列定义之间以逗号分隔，全部列定义都在一对圆括号里（左括号在第一列之前，右括号在最后一列之后）。

注意：不同的实现对数据类型的规定有所不同

不同实现对于名称长度与可使用的字符具有不同的规定。

这个表里的每条记录，也就是每一行数据，会包含以下内容：

```

EMP_ID, EMP_NAME, EMP_ST_ADDR, EMP_CITY, EMP_ST, EMP_ZIP, EMP_PHONE,
EMP_PAGER

```

在这个表里，每个字段就是一列。列EMP\_ID可能包含一个雇员的标识号码，也可能包含多个，这取决于数据库查询或业务的需要。

### 3.3.4 命名规范

在为对象选择名称时，特别是表和列的名称，应该让名称反应出所保存的数据。比如说，保存雇员信息的表可以命名为EMPLOYEE\_TBL。列的名称也是如此，比如保存雇员电话号码的列，显然命名为PHONE\_NUMBER是比较合适的。

### 3.3.5 ALTER TABLE命令

在表被创建之后，我们可以使用ALTER TABLE命令对其进行修改。可以添加列、删除列、修改列定义、添加和去除约束，在某些实现中还可以修改表STORAGE值。ALTER TABLE命令的标准如下所示：

```
alter table table_name [modify] [column column_name][datatype| null not
null]
[restrict| cascade]
[drop]    [constraint constraint_name]
[add]    [column] column definition
```

#### 一、修改表的元素

列的属性是其所包含数据的规则和行为。利用ALTER TABLE命令可以修改列的属性，在此“属性”的含义是：

列的数据类型；

列的长度、有效位数或标度；

列值能否为空。

下面的范例使用ALTER TABLE命令修改表EMPLOYEE\_TBL的EMP\_ID列：

```
ALTER TABLE EMPLOYEE_TBL MODIFY
EMP_ID VARCHAR(10);
Table altered.
```

这一列定义的数据类型没有变，但是长度从9变为10。

## 二、添加列

如果表已经包含数据，这时添加的列就不能定义为NOT NULL，这是一条基本规则。NOT NULL意味着这一列在每条记录里都必须包含数据。所以，在添加一条定义为NOT NULL的列时，如果现有的记录没有包含新列所需要的数据，我们会陷入到自相矛盾的境地。

因此，强行向表添加一列的方法如下：

1. 添加一列，把它定义为NULL（这一行不一定要包含数据）；
2. 给这个新列在每条记录里都插入数据；
3. 把列的定义修改为NOT NULL。

## 三、添加自动增加的列

有时我们需要一列的数据能够自动增加，从而让每一行都具有不同的序号。在很多情况下都需要这样做，比如数据中如果没有适合充当主键的值，或是我们想利用序列号对数据进行排序。创建自动增加的列是相当简单的。MySQL 提供了 SERIAL 方法为表生成真正的唯一值，如下所示：

```
CREATE TABLE TEST_INCREMENT(  
    ID          SERIAL,  
    TEST_NAME   VARCHAR(20));
```

注意：在创建表时使用**NULL**

列的默认属性是NULL，所以在CREATE TABLE语句里不必明确设置。但NOT NULL必须明确指定。

Microsoft SQL Server中可以使用 IDENTITY类型，代码如下：

```
CREATE TABLE TEST_INCREMENT(  
    ID          INT IDENTITY(1,1) NOT NULL,  
    TEST_NAME   VARCHAR(20));
```

Oracle 没有提供直接的方法来创建自动增加的列。但却可以使用 SEQUENCE 对象和一个触发器来实现类似的效果。相关内容将在第22章介绍。

下面，我们可以向新创建的表中插入记录，而不用为自动增加的列指定值：

```
INSERT INTO TEST_INCREMENT(TEST_NAME)
VALUES ('FRED'),('JOE'),('MIKE'),('TED');
```

```
SELECT * FROM TEST_INCREMENT;
```

ID	TEST_NAME
1	FRED
2	JOE
3	MIKE
4	TED

#### 四、修改列

在修改现有表里的列时，需要考虑很多因素。下面是修改列的一些通用规则：

列的长度可以增加到特定数据类型所允许的最大长度；

如果想缩短某列的长度，则必须要求这一列在表里所有数据的长度都小于或等于新长度；

数值数据的位数可以增加；

如果要缩短数值数据的位数，则必须要求这一列在表里所有数值的位数小于或等于新指定的位数；

数值里的小数码数可以增加或减少；

列的数据类型一般是可以改变的。

有些实现会限制用户使用 ALTER TABLE 的某些选项。举例来说，可能不允许从表里撤销列。为了绕过这种限制，我们可以撤销整个表，

然后重建新的表。如果某一列是依赖于其他表的列，或是被其他表的列所引用，在撤销这一列时就可能发生问题。详细情况请查看具体实现的文档。

注意：创建练习表

在本章后面的练习里会创建这些表。在第5章里会向这些表填充数据。

### [3.3.6 从现有表新建另一个表](#)

警告：修改或删除表时务必小心

在修改或删除表时一定要小心。如果在发布这些命令时出现逻辑或输入错误，就可能导致丢失重要数据。

利用CREATE TABLE语句与SELECT语句的组合可以复制现有的表。新表具有同样的列定义，我们可以选择任何列或全部列。由函数或多列组合创建出来的列会自动保持数据所需的大小。从另一个表创建新表的基本语法如下所示：

```
create table new_table_name as
select [ * | column1, column2 ]
from table_name
[ where ]
```

注意其中的一些新关键字，特别是SELECT。SELECT是数据库查询语句，将在第7章详细介绍。现在需要掌握的就是我们可以利用查询的结果创建一个表。

MySQL和Oracle都支持使用CREATE TABLE AS SELECT方法，在一个表的基础上创建另一个表。但是Microsoft SQL Server却不一样，它使用SELECT...INTO方法来实现相同的效果。示例如下所示：

```
select [ * | column1, columnn2]
into new_table_name
from table_name
[ where ]
```

下面有一些示例使用了这种方法。

首先，我们进行一个简单的查询来了解表PRODUCTS\_TBL里的内容。

```
select * from products_tbl;
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95

接下来，基于前面这个查询创建名为PRODUCTS\_TMP的表：

```
create table products_tmp as
select * from products_tbl;
```

Table created.

在SQL Server中，需要使用如下命令：

```
select *
into products_tmp
from products_tbl;
```

Table created.



现在，如果对表PRODUCTS\_TMP进行查询，得到的数据与原始表是一样的。

```
select *
from products_tmp;
PROD_ID      PROD_DESC                                COST
-----
11235        WITCH COSTUME                                29.99
222          PLASTIC PUMPKIN 18 INCH                     7.75
13           FALSE PARAFFIN TEETH                      1.1
90           LIGHTED LANTERNS                       14.5
15           ASSORTED COSTUMES                           10
9            CANDY CORN                          1.35
6            PUMPKIN CANDY                          1.45
87           PLASTIC SPIDERS                      1.05
119          ASSORTED MASKS                       4.95
```

注意：“\*”的意义

SELECT \*会选择指定表里全部字段的数据。“\*”表示表里的一行完整数据，也就是一条完整记录。

注意：默认使用相同的**STORAGE**属性

从现有表创建新表，新表与原始表具有一样的属性。

### [3.3.7 删除表](#)

删除表是一种相当简单的操作。如果使用了RESTRICT选项，并且表被视图或约束所引用，DROP语句就会返回一个错误。当使用了CASCADE选项时，删除操作会成功执行，而且全部引用视图和约束都被删除。删除表的语法如下所示：

```
drop table table_name [ restrict|cascade ]
```

在SQL Server中，不能使用CASCADE选项。因此，要在SQL Server中删除表，必须同时删除与该表有引用关系的所有对象，以避免系统中

遗留无效对象。

下面这个范例删除刚才创建的表：

```
drop table products_tmp;
```

```
Table dropped.
```

警告：删除表的操作务必指向准确

在删除表时，在提交命令之前要确保指定了表的规划名或所有者，否则可能误删除其他的表。如果使用多用户账户，在删除表之前一定要确定使用了适当的用户名连接数据库。

## [3.4 完整性约束](#)

完整性约束用于确定关系型数据库里数据的准确性和一致性。在关系型数据库里，数据完整性是通过引用完整性的概念实现的，而在引用完整性里包含了很多类型。

### [3.4.1 主键约束](#)

主键是表里一个或多个用于实现记录唯一性的字段。虽然主键通常是由一个字段构成的，但也可以由多个字段组成。举例来说，雇员的社会保险号码或雇员被分配的标识号码都可以在雇员表里作为主键。主键的作用在于表里每条记录都具有唯一的值。由于在雇员表里一般不会出现用多条记录表示一个雇员的情况，所以雇员的标识号码可以作为主键。主键是在创建表时指定的。

下面的范例把字段EMP\_ID指定为表EMPLOYEES\_TBL的主键（PRIMARY KEY）：

```

CREATE TABLE EMPLOYEE_TBL
(EMP_ID          CHAR(9)          NOT NULL PRIMARY KEY,
EMP_NAME         VARCHAR (40)     NOT NULL,
EMP_ST_ADDR      VARCHAR (20)     NOT NULL,
EMP_CITY         VARCHAR (15)     NOT NULL,
EMP_ST          CHAR(2)           NOT NULL,
EMP_ZIP          INTEGER(5)       NOT NULL,
EMP_PHONE        INTEGER(10)      NULL,
EMP_PAGER        INTEGER(10)      NULL);

```

这种定义主键的方法是在创建表的过程中完成的，这时主键是个隐含约束。我们还可以在创建表时明确地指定主键作为一个约束，如下所示：

```

CREATE TABLE EMPLOYEE_TBL
(EMP_ID          CHAR(9)          NOT NULL,
EMP_NAME         VARCHAR (40)     NOT NULL,
EMP_ST_ADDR      VARCHAR (20)     NOT NULL,
EMP_CITY         VARCHAR (15)     NOT NULL,
EMP_ST          CHAR(2)           NOT NULL,
EMP_ZIP          INTEGER(5)       NOT NULL,
EMP_PHONE        INTEGER(10)      NULL,
EMP_PAGER        INTEGER(10)      NULL,
PRIMARY KEY (EMP_ID));

```

在这个范例里，主键约束是在CREATE TABLE语句里的字段列表之后定义的。

包含多个字段的主键可以用如下两种方法之一来定义，以下示例适用于Oracle数据库：

```

CREATE TABLE PRODUCT_TST
(PROD_ID          VARCHAR2(10)          NOT NULL,
 VENDOR_ID        VARCHAR2(10)          NOT NULL,
 PRODUCT          VARCHAR2(30)          NOT NULL,
 COST             NUMBER(8,2)           NOT NULL,
 PRIMARY KEY (PROD_ID, VENDOR_ID));

ALTER TABLE PRODUCTS_TST
ADD CONSTRAINT PRODUCTS_PK PRIMARY KEY (PROD_ID, VENDOR_ID);

```

### [3.4.2 唯一性约束](#)

唯一性约束要求表里某个字段的值在每条记录里都是唯一的，这一点与主键类似。即使我们对一个字段设置了主键约束，也可以对另一个字段设置唯一性约束，尽管它不会被当作主键使用。

研究下面这个范例：

```

CREATE TABLE EMPLOYEE_TBL
(EMP_ID           CHAR(9)               NOT NULL      PRIMARY KEY,
 EMP_NAME         VARCHAR (40)          NOT NULL,
 EMP_ST_ADDR      VARCHAR (20)          NOT NULL,
 EMP_CITY         VARCHAR (15)          NOT NULL,
 EMP_ST           CHAR(2)               NOT NULL,
 EMP_ZIP          INTEGER(5)            NOT NULL,
 EMP_PHONE        INTEGER(10)           NULL          UNIQUE,
 EMP_PAGER        INTEGER(10)           NULL);

```

在这个范例里，主键是EMP\_ID字段，表示雇员标识号码，用于确保表里的每条记录都是唯一的。主键通常是在查询里引用的字段，特别是用于结合表时。字段EMP\_PHONE也会定义为UNIQUE，表示任意两个雇员都不能有相同的电话号码。这两个都具有唯一性的字段之间没有太多的区别，只是主键让表具有了一定的秩序，并且可以用于结合相互关联的表。

### [3.4.3 外键约束](#)

外键是子表里的一个字段，引用父表里的主键。外键约束是确保表与表之间引用完整性的主要机制。一个被定义为外键的字段用于引用另一个表里的主键。

研究下面范例里外键的创建：

```
CREATE TABLE EMPLOYEE_PAY_TST
(EMP_ID          CHAR(9)          NOT NULL,
 POSITION        VARCHAR2(15)     NOT NULL,
 DATE_HIRE       DATE             NULL,
 PAY_RATE        NUMBER(4,2)      NOT NULL,
 DATE_LAST_RAISE DATE            NULL,
 CONSTRAINT EMP_ID_FK FOREIGN KEY (EMP_ID) REFERENCES EMPLOYEE_TBL
 (EMP_ID));
```

在这个范例里，EMP\_ID 字段被定义为表 EMPLOYEE\_PAY\_TBL 的外键，它引用了表EMPLOYEE\_TBL 里的 EMP\_ID 字段。这个外键确保了表 EMPLOYEE\_PAY\_TBL 里的每个EMP\_ID 都在表 EMPLOYEE\_TBL 里有对应的 EMP\_ID。这被称为父/子关系，其中父表是EMPLOYEE\_TBL，子表是EMPLOYEE\_PAY\_TBL。请观察表3.4来更好地理解父子表的关系。

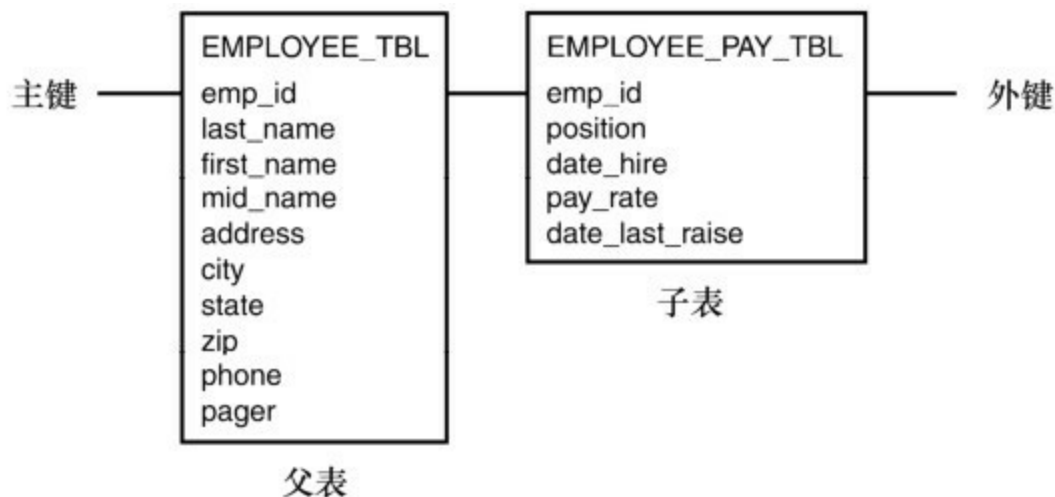


图3.4 父/子表关系

在这个图里，子表里的EMP\_ID字段引用父表里的EMP\_ID字段。为了在子表里插入一个EMP\_ID的值，它首先要存在于父表的EMP\_ID里。类似地，父表里删除一个EMP\_ID的值，子表里相应的EMP\_ID值必须全部被删除。这就是引用完整性的概念。

利用ALTER TABLE命令可以向表里添加外键，比如下面这个范例：

```
alter table employee_pay_tbl
add constraint id_fk foreign key (emp_id)
references employee_tbl (emp_id);
```

注意：ALTER TABLE命令在不同的SQL实现中有所不同

ALTER TABLE命令的选项在不同SQL实现里是不同的，特别是关于约束的选项。另外，约束的实际使用与定义也有所不同，但引用完整性的概念在任何关系型数据库里都是一样的。

### [3.4.4 NOT NULL约束](#)

前面的范例在每个字段的数据类型之后使用了关键字NULL和NOT NULL。NOT NULL也是一个可以用于字段的约束，它不允许字段包含NULL值；换句话说，定义为NOT NULL的字段在每条记录里都必须有值。在没有指定NOT NULL时，字段默认为NULL，也就是可以是NULL值。

### [3.4.5 检查约束](#)

检查（CHK）约束用于检查输入到特定字段的数据的有效性，可以提供后端的数据库编辑，虽然编辑通常是在前端程序里完成的。一般情况下，编辑功能限制了能够输入到字段或对象的值，无论这个功能是在数据库还是在前端程序里实现的。检查约束为数据提供了另一层保护。

下面的范例展示了在Oracle中，检查约束的使用：

```
CREATE TABLE EMPLOYEE_CHECK_TST
(EMP_ID          CHAR(9)          NOT NULL,
EMP_NAME         VARCHAR2(40)     NOT NULL,
EMP_ST_ADDR      VARCHAR2(20)     NOT NULL,
EMP_CITY         VARCHAR2(15)     NOT NULL,
EMP_ST          CHAR(2)           NOT NULL,
EMP_ZIP          NUMBER(5)        NOT NULL,
EMP_PHONE        NUMBER(10)       NULL,
EMP_PAGER        NUMBER(10)       NULL,
PRIMARY KEY (EMP_ID),
CONSTRAINT CHK_EMP_ZIP CHECK ( EMP_ZIP = '46234' ));
```

表里的 EMP\_ZIP 字段设置了检查约束，确保了输入到这个表里的全部雇员的 ZIP 代码都是“46234”。虽然这显得有些过于严格，但这不要紧，足以展示如何使用检查约束了。

如果想利用检查约束来确保ZIP代码属于某个值列表，可以像下面这样使用检查约束：

```
CONSTRAINT CHK_EMP_ZIP CHECK ( EMP_ZIP in ( '46234','46227','46745' ) );
```

如果想指定雇员的最低小时工资，可以像下面这样设置约束：

```
CREATE TABLE EMPLOYEE_PAY_TBL
(EMP_ID          CHAR(9)          NOT NULL,
POSITION         VARCHAR2(15)     NOT NULL,
DATE_HIRE        DATE            NULL,
PAY_RATE         NUMBER(4,2)      NOT NULL,
DATE_LAST_RAISE  DATE            NULL,
CONSTRAINT EMP_ID_FK FOREIGN KEY (EMP_ID) REFERENCES EMPLOYEE_TBL
(EMP_ID),
CONSTRAINT CHK_PAY CHECK ( PAY_RATE > 12.50 ) );
```

在这个范例里，表里的任何雇员的小时工资都不能低于\$12.50。在检查约束里可以使用几乎任何条件，就像在SQL查询里一样。第5章和第7章将更详细地介绍这些条件。

### [3.4.6 去除约束](#)



利用ALTER TABLE命令的DROP CONSTRAINT选项可以去除已经定义的约束。举例来说，如果想去除表EMPLOYEES里的主键约束，可以使用下面的命令：

```
ALTER TABLE EMPLOYEES DROP CONSTRAINT EMPLOYEES_PK;
```

```
Table altered.
```

有些SQL实现还提供了去除特定约束的快捷方式。举例来说，在MySQL里可以使用下面这样的命令来去除主键约束：

```
ALTER TABLE EMPLOYEES DROP PRIMARY KEY;
```

```
Table altered.
```

注意：有些实现允许中止约束，这样我们可以选择暂时中止它，而不是从数据库里去除它，稍后还可以再启动它。

### [3.5 小结](#)

本章概述了数据库对象的基本知识，主要介绍了表。表是关系型数据库里最简单的数据存储方式，它包含成组的逻辑信息，比如雇员、顾客或产品信息。表由各种字段组成，每个字段都有自己的属性，主要包括数据类型和约束，比如NOT NULL、主键、外键和唯一值。

本章介绍了 CREATE TABLE命令和选项，比如存储参数。还介绍了如何使用 ALTER TABLE命令调整已有表的结构。虽然管理数据库表的过程并不是SQL里最基本的过程，但如果首先学习了表的结构与本质，我们就能更容易地掌握通过数据操作或数据库查询来访问表的概念。下一章将介绍SQL对其他对象的管理，比如表的索引和视图。

### [3.6 问与答](#)



问：在创建表的过程中给表命名时，一定要使用像**\_TBL**这样的后缀吗？

答：当然不是。没有规定必须使用。举例来说，保存雇员信息的表可以具有下面这些名称，或是任何能够说明表里保存了何种数据的名称：

```
EMPLOYEE  
EMP_TBL  
EMPLOYEE_TBL  
EMPLOYEE_TABLE  
WORKER
```

问：在删除表时，为什么使用规划名称是非常重要的？

答：有一个 **DBA** 新手删除表的真实故事：一个程序员在他的规划下创建了一个表，其名称与一个产品表是一样的。这名程序员后来离开了公司，他在数据库里的账户也要被删除，但**DROP USER**命令报告出错，因为还有他拥有的对象没有被删除。在经过一些调查之后，这名程序员创建的表被认定是没有用的，于是就要使用**DROP TABLE**命令了。

问题在于当执行**DROP TABLE**命令时，**DBA**以产品规划登录到数据库。这名**DBA**在删除表时，应该指定规划名称或所有者，但是他没有，结果是删除了另一个规划里不该删除的表。而恢复这个表花掉了大约8个小时。

## [3.7 实践](#)

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习是为了把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### [3.7.1 测验](#)

1. 下面这个CREATE TABLE命令能够正常执行吗？需要做什么修改？在不同的数据库（MySQL、Oracle、SQL Server）中执行，有什么限制吗？

```
Create table EMPLOYEE_TABLE as:
( ssn          number(9)          not null,
  last_name     varchar2(20)       not null,
  first_name    varchar2(20)       not null,
  middle_name   varchar2(20)       not null,
  st address    varchar2(30)       not null,
  city          char(20)           not null,
  state         char(2)            not null,
  zip           number(4)          not null,
  date hired    date);
```

2. 能从表里删除一个字段吗？
3. 在前面的表EMPLOYEE\_TBL里创建一个主键约束应该使用什么语句？
4. 为了让前面的表EMPLOYEE\_TBL里的MIDDLE\_NAME字段可以接受NULL值，应该使用什么语句？
5. 为了让前面的表 EMPLOYEE\_TBL 里添加的人员记录只能位于纽约州('NY')，应该使用什么语句？
6. 要在前面的表EMPLOYEE\_TBL里添加一个名为EMPID的自动增量字段，应该使用什么语句，才能同时符合MySQL和SQL Server的语法结构？

### 3.7.2 练习

通过下面的练习，读者将创建出数据库中所有的表，以便为后续章节提供练习环境。此外，还需要执行一些命令，来检查表结构。为了确保准确无误，我们分别介绍三种数据库实现（MySQL、Microsoft SQL Server、Oracle）的操作方法，这些方法在具体的实现上会有微小的差

异。

## Mysql

打开命令行窗口，使用下面的命令语法登录到本地的 MySQL，用实际的用户名替换username，用实际的密码替换password。注意在-p与密码之间没有空格。

```
Mysql -h localhost -u username -ppassword
```

在mysql>提示符下，输入以下命令，告诉MySQL我们要使用哪个数据库：

```
use learnsql;
```

现在转到附录 D 来了解本书中所使用的 DDL。在 mysql>提示符下输入每个 CREATE TABLE命令，注意要包含每个命令之后的分号。这些命令创建的表将用于整本书的学习。

在mysql>提示符下，输入以下命令来列出所有的表：

```
show tables;
```

在mysql>提示符下，使用DESCRIBE命令（缩写为desc）列出每个表的全部字段和它们的属性，如下所示：

```
describe employee_tbl;  
describe employee_pay_tbl;
```

如果遇到任何错误提示或输入错误，只需要重新创建相应的表即可。如果表成功创建了，但有输入错误（比如没有正确地定义字段，或是漏掉了某个字段），就删除表，再使用CREATE TABLE命令。DROP TABLE命令的语法如下所示：

```
drop table orders_tbl;
```

## Microsoft SQL Server

打开命令行窗口，使用下面的命令语法登录到本地的SQL Server，用实际的用户名替换username，用实际的密码替换password。注意在-p与密码之间没有空格。

```
SQLCMD -S localhost -U username -Ppassword
```

在1>提示符下，输入以下命令，告诉SQL Server我们要使用哪个数据库。在使用SQLCMD时，需要用关键字GO来执行输入的命令。

```
1>use learnsql;  
2>GO
```

现在转到附录D来了解本书中所使用的DDL。在1>提示符下输入每个CREATE TABLE命令，注意要包含每个命令之后的分号，最后还要使用关键字GO来执行命令。这些命令创建的表将用于整本书的学习。

在1>提示符下，输入以下命令来列出所有的表。在命令后面加上关键字GO来执行：

```
Select name from sys.tables;
```

在1>提示符下，使用存储过程sp\_help列出每个表的全部字段和它们的属性，如下所示：

```
Sp_help_employee_tbl;  
Sp_help_employee_pay_tbl;
```

如果遇到任何错误提示或输入错误，只需要重新创建相应的表即可。如果表成功创建了，但有输入错误（比如没有正确地定义字段，或

是漏掉了某个字段），就删除表，再使用CREATE TABLE命令。DROP TABLE命令的语法如下所示：

```
drop table orders_tbl;
```

## Oracle

打开命令行窗口，使用下面的命令语法登录到本地的Oracle。输入用户名和密码。

```
sqlplus
```

现在转到附录 D 来了解本书中所使用的 DDL。在 SQL>提示符下输入每个 CREATE TABLE命令，注意要包含每个命令之后的分号。这些命令创建的表将用于整本书的学习。

在SQL>提示符下，输入以下命令来列出所有的表：

```
Select * from cat;
```

在SQL>提示符下，使用DESCRIBE命令（缩写为desc）列出每个表的全部字段和它们的属性，如下所示：

```
describe employee_tbl;  
describe employee_pay_tbl;
```

如果遇到任何错误提示或输入错误，只需要重新创建相应的表即可。如果表成功创建了，但有输入错误（比如没有正确地定义字段，或是漏掉了某个字段），就删除表，再使用CREATE TABLE命令。DROP TABLE命令的语法如下所示：

```
drop table orders_tbl;
```

## [第4章 规格化过程](#)

本章的重点包括：

什么是规格化

规格化的优点

去规格化的优点

规格化技术

规格化的方针

数据库设计

本章介绍把原始数据库分解为表的过程，这被称为规格化。数据库开发人员利用规格化过程来设计数据库，使其更便于组织和管理，同时确保数据在整个数据库里的正确性。这一过程在各种RDBMS中都是一样的。

本章会介绍规格化与去规格化的优缺点，以及规格化带来的数据完整性与性能之间的矛盾。

### [4.1 规格化数据库](#)

规格化是去除数据库里冗余数据的过程，在设计和重新设计数据库时使用。它是一组减少数据冗余来优化数据库的指导方针，具体的方针被称为规格形式，稍后将详细介绍。在本书中是否应该包含介绍规格化的内容是个两难的决定，因为其规则对于 SQL 初学者来说过于复杂了。然而，规格化是个十分重要的过程，对它的理解会加深我们对 SQL 的掌握。本章尽量简化对规格化的介绍，不会过于关注规格化的细节，而且着重于让读者理解其基本概念。

#### [4.1.1 原始数据库](#)

在没有经过规格化的数据库里，有些数据可能会出现在多个不同的

表里，而且没有什么明显的原因。这样对安全、磁盘利用、查询速度、数据库更新都不好，特别是可能产生数据完整性的问题。在规格化之前，数据库里的数据并没有从逻辑上被分解到较小的、更易于管理的表里，图4.1展示了本书所使用的数据库在规格化之前的状态。

COMPANY_DATABASE	
emp_id	cust_id
last_name	cust_name
first_name	cust_address
middle_name	cust_city
address	cust_state
city	cust_zip
state	cust_phone
zip	cust_fax
phone	ord_num
pager	qty
position	ord_date
date_hire	prod_id
pay_rate	prod_desc
bonus	cost
date_last_raise	

图4.1 原始数据库

在数据库逻辑设计过程中，确定原始数据库里的信息由什么组成是第一个也是最重要的步骤，我们必须了解组成数据库的全部数据元素，才能有效地使用规格化技术。只有用必要的时间收集所需的数据集，才能避免因为丢失数据元素而重新设计数据库。

#### 4.1.2 数据库逻辑设计

任何数据库设计都要考虑到终端用户。数据库逻辑设计，也被称为逻辑建模，是把数据安排到逻辑的、有组织的对象组，以便于维护的过程。数据库的逻辑设计应该减少数据重复，甚至是完全消除这种现象。

毕竟，为什么要把数据存储两遍呢？另外，数据库逻辑设计应该努力让数据库易于维护和更新，同时也要保持数据库里的命名规范与逻辑。

### 一、什么是终端用户的需求

在设计数据库时，终端用户的需求应该是最重要的考虑因素。记住，终端用户是最终使用数据库的人。利用用户的前端工具（允许用户访问数据库的客户程序），数据库的使用应该是相当简单的，但是在设计数据库时如果没有考虑到用户的需求，这也许就不能达到。性能优化也是如此。

在设计时要考虑的与用户相关的因素包括：

数据库里应该保存什么数据？

用户如何访问数据库？

用户需要什么权限？

数据库里的数据如何分组？

哪些数据最经常被访问？

全部数据与数据库如何关联？

采取什么措施保证数据的正确性？

采取什么措施减少数据冗余？

采取什么措施让负责维护数据的用户更易于使用数据库？

### 二、数据冗余

数据应该没有冗余；这意味着重复的数据应该保持到最少，其原因有很多。举例来说，把雇员的家庭住址保存到多个表里就没有意义。重复数据会占据额外的存储空间，而且经常会产生混乱，比如如果雇员的地址在一个表里的内容与在另一个表里的内容不相符时，哪一个是正确的呢？能不能找到文档资料来确定雇员当前的地址？数据管理已经很困难了，冗余数据会导致灾难。减少冗余数据还能简化数据库的更新操作。如果只有一个表保存了雇员的地址，那么在用新地址更新了这个表之后，我们就可以确保所有人都会看到这个更新的数据。



### 4.1.3 规格形式

这一章讨论规格形式，这是数据库规格化过程中必不可少的一个概念。

规格形式是衡量数据库被规格化级别（或深度）的一种方式。数据库的规格化级别是由规格形式决定的。

下面是规格化过程中最常见的3种规格形式：

第一规格形式；

第二规格形式；

第三规格形式。

除此之外，还有其他规格形式，但都不常用。在这3种主要的规格形式中，每一种都依赖于前一种形式所采用的规格化步骤。举例来说，如果想以第二规格形式对数据库进行规格化，数据库必须处于第一种规格形式。

#### 一、第一规格形式

第一规格形式的目标是把原始数据分解到表中。在所有表都设计完成之后，给大多数表或全部表设置一个主键。从第3章中可以知道，主键必须是个唯一的值，所以在选择主键时应该尽量选择能够从本质上唯一区别数据的元素。图4.2展示了图4.1所示原始数据库使用第一规格形式重新设计之后的情况。

从图中可以看出，为了达到第一规格形式，数据被分解为包含相关信息的逻辑单元，每个逻辑单元都有一个主键，而且任何表里都没有重复的数据组。现在的数据库不再是一个大表，而是被分解为较小的、更易于管理的表：EMPLOYEE\_TBL、CUSTOMER\_TBL 和 PRODUCTS\_TBL。主键通常是表里的第一列，本例中分别是 EMP\_ID、CUST\_ID和PROD\_ID。这种命名方式是在设计数据库时常用的规范，确保了各种名称的可读性。

主键也可以由表中的多个列构成。这类主键所涉及的数据通常不是数据库自动生成的数字，而是有逻辑意义的数据，例如生产商的名称或者一本书的 ISBN 编号。这类数据被称为自然主键，即使不在数据库中，也可以通过它们来区分不同的对象。在为表选择主键的时候，需要注意的一点就是，主键必须能够唯一地定义表中的一条记录。否则，查询的结果可能会返回重复的记录，而且也无法通过主键来删除一条特定的记录。

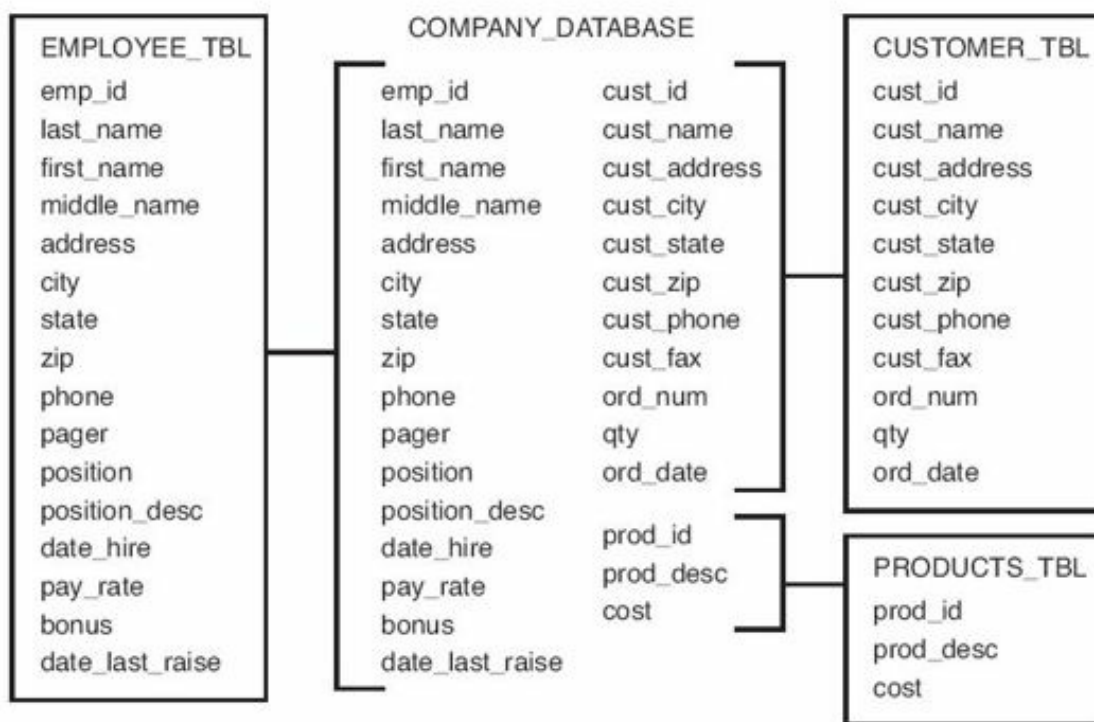


图4.2 第一规格形式

## 二、第二规格形式

第二规格形式的目标是提取对主键仅有部分依赖的数据，把它们保存到另一个表里。图4.3展示了第二规格形式。

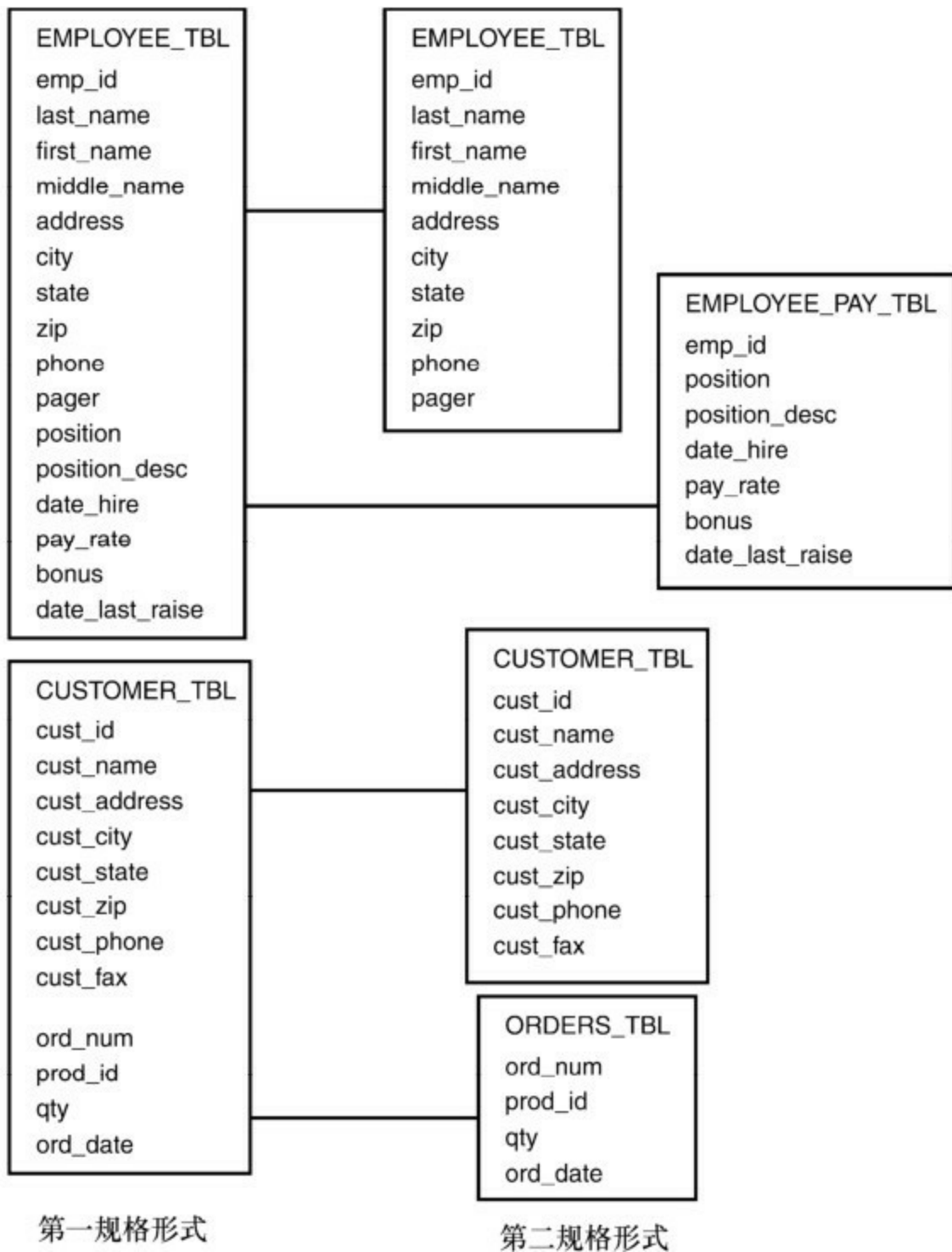


图4.3 第二规格形式

从图中可以看出，第二规格形式以第一规格形式为基础，把两个表

进一步划分为更明确的单元。

EMPLOYEE\_TBL被分解为两个表，分别是EMPLOYEE\_TBL和EMPLOYEE\_PAY\_TBL。雇员个人信息是依赖于主键（EMP\_ID）的，保留在EMPLOYEE\_TBL表里的都是如此（EMP\_ID、LAST\_NAME、FIRST\_NAME、MIDDLE\_NAME、ADDRESS、CITY、STATE、ZIP、PHONE和PAGER）。而在另一方面，与EMP\_ID仅部分依赖的信息被转移到EMPLOYEE\_PAY\_TBL（包括EMP\_ID、POSITION、POSITION\_DESC、DATE\_HIRE、PAY\_RATE和DATE\_LAST\_RAISE）。注意到两个表都包含列EMP\_ID，这是每个表的主键，用于在两个表之间匹配对应的数据。

CUSTOMER\_TBL被分解为两个表，分别是CUSTOMER\_TBL和ORDERS\_TBL，具体情况类似于EMPLOYEE\_TBL，仅部分依赖于主键的列被转移到另一个表。顾客的订单信息依赖于每一个CUST\_ID，但与顾客的一般信息没有直接依赖关系。

### 三、第三规格形式

第三规格形式的目标是删除表里不依赖于主键的数据。图4.4展示了第三规格形式。

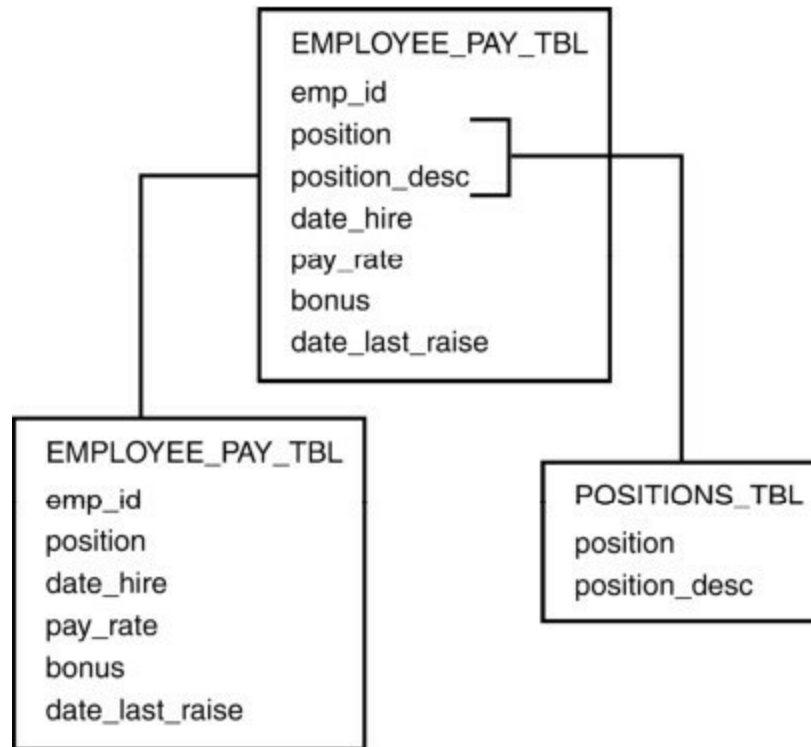


图4.4 第三规格形式

这里又创建了一个新表来实现第三规格形式。**EMPLOYEE\_PAY\_TBL**被分解为两个表：一个表保存雇员的实际支付信息，另一个表保存职位描述。这的确不需要保存在**EMPLOYEE\_PAY\_TBL**里，列**POSITION\_DESC**与主键**EMP\_ID**完全不相干。从上述介绍可以看出，规格化过程就是采取一系列步骤，把原始数据分解为由关联数据形成的多个表。

#### 4.1.4 命名规范

命名规范是在数据库规格化过程中最重要的考虑因素之一。名称是我们引用数据库对象的方式。表的名称应该能够描述所保存信息的类型，以便于我们找到需要的数据。对于没有参加数据库设计而需要查询数据库的用户来说，具有描述性的名称更为重要。

应该在公司范围内统一命名规范，不仅是数据库里表的命名，而是

用户、文件和其他相关对象的命名都应该遵守。命名规范还让我们更容易判断表的用途和数据库系统里文件的位置，从而有助于数据库管理。设计和坚持命名规范是公司开发成功数据库实现的第一步。

#### 4.1.5 规格化的优点

规格化为数据库带来了很多好处，主要包括以下几点：

更好的数据库整体组织性；

减少冗余数据；

数据库内部的数据一致性；

更灵活的数据库设计；

更好地处理数据库安全；

加强引用整体性的概念。

组织性是由规格化过程所产生的，让从访问数据库的用户到负责管理数据库所有对象的管理员（DBA）的所有人都感到更轻松。数据冗余被减少了，从而简化了数据结构，节约了磁盘空间。由于重复数据被尽量减少了，所以数据不一致的可能大大降低。举例来说，某人在一个表的姓名可能是STEVE SMITH，而在另一个表里是STEPHEN R. SMITH。减少重复数据提高了数据完整性，或者说数据库里数据的一致性和准确性。数据库规格化之后，分解为较小的表，便于我们更灵活地修改现有的结构。显然，修改包含较少数据的小表，要比修改包含数据库全部重要数据的一个大表要轻松得多。最后，DBA能够控制特定用户对特定表的访问，从而提高了安全性。在进行了规格化之后，安全就更容易控制了。

引用完整性表示一个表里某列的值依赖于另一个表里某列的值。举例来说，如果某个顾客要在表ORDERS\_TBL里有一条记录，则必须首先在表CUSTOMER\_TBL里有一条记录。完整性约束还可以限制列的取值范围，它应该在创建表时设置。引用完整性一般是通过使用主键和外

键来控制的。

在一个表里，外键（通常是一个字段）直接引用另一个表里的主键来实现引用完整性。在前一个图里，表ORDERS\_TBL里的CUST\_ID就是一个外键，它引用表CUSTOMER\_TBL里的CUST\_ID。规格化过程把数据从逻辑上分解为由主键引用的子集，从而有助于加强和坚持这些约束。

#### [4.1.6 规格化的缺点](#)

虽然大多数成功的数据库都在一定程度上进行了规格化，但规格化的确有一个不可回避的缺点：降低数据库性能。性能降低的程度取决于查询或事务被提交给数据库的时机，其中涉及多个因素，比如CPU使用率、内存使用率和输入/输出（I/O）。简单来说，规格化的数据库比非规格化的数据库需要更多的CPU、内存和I/O来处理事务和查询。规格化的数据库必须找到所需的表，然后把这些表的数据结合起来，从而得到需要的信息或处理相应的数据。关于数据库性能的更详细讨论请见第18章。

### [4.2 去规格化数据库](#)

去规格化是修改规格化数据库的表的构成，在可控制的数据冗余范围内提高数据库性能。尝试提高性能是进行去规格化数据库的唯一原因。去规格化的数据库与没有进行规格化的数据库不一样，去规格化是在数据库规格化基础上进行一些调整，因为规格化的数据库需要频繁地进行表的结合而降低了性能（关于表的结合请见第13章）。去规格化会把一些独立的表合成在一起，或是创建重复的数据，从而减少在数据检索时需要结合的表的数量，进而减少所需的I/O和CPU时间。这在较大的数据仓库程序中会有明显的好处，其中的计算可能会涉及表里数以百万行的数据。



去规格化也是有代价的。它增加了数据冗余，虽然提高了性能，但需要付出更多的精力来处理相关的数据。程序代码会更加复杂，因为数据被分散到多个表，而且可能更难于定位。另外，引用完整性更加琐碎，因为相关数据存在于多个表里。规格化与去规格化都有好处，但都需要我们对实际的数据和公司的详细业务需求有全面的了解。在确定要着手进行去规格化时，一定要仔细记录所采取的过程，以便于更好地处理像数据冗余这样的问题，维护系统内部的数据完整性。

### [4.3 小结](#)

在进行数据库设计时，必须做出一个困难的决定——规格化或去规格化，这的确是个问题。一般来说，数据库问题需要进行一定程度的规格化，但到什么程度才不至于严重影响性能呢？答案取决于程序本身。数据库有多大？其用途是什么？什么样的用户要访问数据？本章介绍了3种最常见的规格形式、规格化过程的底层概念、数据的完整性。规格化过程包含多个步骤，大多数都不是必需的，但对于数据库的功能和性能来说都是很重要的。无论决定进行什么程度的规格化，总是会存在便于维护与性能降低，或复杂维护与更好性能之间的平衡。最终，设计数据库的个人（或团队）必须做出决定，并对此负责。

### [4.4 问与答](#)

问：在设计数据库时为什么要考虑最终用户的需求？

答：最终用户才是真正使用数据库的人，从这种角度来说，他们应该是任何数据库设计的中心。数据库设计人员只不过是帮助组织数据而已。

问：规格化要比去规格化好吗？

答：可能是这样的，但是到达一定程度时，去规格化可能会更好，



这其中受到很多因素的影响。我们会对数据库进行规格化来减少其中的重复数据，到达一定程度之后可能又会转回头来，通过去规格化来改善性能。

## 4.5 实践

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习是为了把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### 4.5.1 测验

1. 判断正误：规格化是把数据划分为逻辑相关组的过程。
2. 判断正误：让数据库里没有重复或冗余数据，让数据库里所有内容都规格化，总是最好的方式。
3. 判断正误：如果数据是第三规格形式，它会自动属于第一和第二规格形式。
4. 与规格化数据库相比，去规格化数据库的主要优点是什么？
5. 去规格化的主要缺点是什么？
6. 在对数据库进行规格化时，如何决定数据是否需要转移到单独的表？
7. 对数据库设计进行过度规格化的缺点是什么？

### 4.5.2 练习

1. 为一家小公司开发一个新数据库，使用如下数据，对其进行规格化。记住，即使是一家小公司，其数据库的复杂程度也会超过这里给出的范例。

雇员：

Angela Smith, secretary, 317-545-6789, RR 1 Box 73, Greensburg, Indiana, 47890, \$9.50 per hour, date started January 22, 2006, SSN is 323149669.

Jack Lee Nelson, salesman, 3334 N. Main St., Brownsburg, IN, 45687, 317-852-9901, salary of \$35,000.00 per year, SSN is 312567342, date started 10/28/2005.

顾客:

Robert's Games and Things, 5612 Lafayette Rd., Indianapolis, IN, 46224, 317-291-7888, customer ID is 432A.

Reed's Dairy Bar, 4556 W 10th St., Indianapolis, IN, 46245, 317-271-9823, customer ID is 117A.

顾客订单:

Customer ID is 117A, date of last order is December 20, 2009, the product ordered was napkins, and the product ID is 661.

2. 像第 3 章介绍的那样登录到你新建的数据库。可以输入以下命令来确保使用的是learnsql数据库:

Use learnsql;

在Oracle中, 这条命令意味着进入规划。默认情况下, 用户在自己的规划中创建对象。

进入数据库后, 根据练习 1中定义的信息, 用CREATE TABLE命令创建相应的表。

## [第5章 操作数据](#)

本章的重点包括：

数据操作语言概述

介绍如何操作表里的数据

数据填充背后的概念

如何从表里删除数据

如何修改表里的数据

本章介绍SQL里的数据操作语言（DML），它用于修改关系型数据库里的数据和表。

## [5.1 数据操作概述](#)

数据操作语言使数据库用户能够对关系型数据库里的数据进行修改，包括用新数据填充表、更新现有表里的数据、删除表里的数据。利用DML命令还可以进行简单的数据库查询。

SQL里3个基本的DML命令是：

INSERT;

UPDATE;

DELETE。

SELECT命令可以与DML命令配合使用，将在第7章详细介绍。SELECT命令是基本的查询命令，在INSERT命令把数据输入到数据库之后使用。所以在本章中，我们会向表中插入数据，以便能够更好地使用SELECT命令。

## [5.2 用新数据填充表](#)

用数据填充表就是把新数据输入到表的过程，无论是使用单个命令的手工过程，还是使用程序或其他相关软件的批处理过程。手工数据填充是指通过键盘输入数据，自动填充通常是从外部数据源（比如其他数

数据库或一个平面文件）获得数据，再把得到的数据加载到数据库。

在用数据填充表时，有很多因素会影响什么数据以及多少数据可以输入到表里。主要因素包括现有的表约束、表的物理尺寸、列的数据类型、列的长度和其他完整性约束（比如主键和外键）。下面将介绍向表输入新数据的基本知识，并且说明什么是可以做的，而什么是不能做的。

### [5.2.1 把数据插入到表](#)

INSERT语句可以把数据插入到表，它具有一些选项，其基本语法如下所示：

```
INSERT INTO TABLE_NAME  
VALUES ('value1', 'value2', [ NULL ] );
```

注意：数据是区分大小写的

不要忘了SQL语句无所谓是大写的或小写的，而数据永远都是区分大小写的。举例来说，如果数据以大写方式输入到数据库，它就必须以大写方式被引用。这些范例使用了大写和小写字母，只是为了展示这样做并不影响结果。

在使用这种语法时，必须在VALUES列表里包含表里的每个列。在这个列表里，每个值之间是以逗号分隔的。字符、日期和时间数据类型的值必须以单引号包围，而数值或NULL值就不必了。表里的每一列都应该有值，并且值的顺序与列在表里的次序一致。在后续章节中，我们会介绍如何指定列的顺序。但是现阶段，读者只需要知道SQL会默认用户在插入数据的时候，使用的是与创建列时相同的顺序。

下面的范例将把一条新记录插入到表PRODUCTS\_TBL里。

表的结构如下所示：

products\_tbl

COLUMN Name	Null?	DATA Type
PROD_ID	NOT NULL	VARCHAR(10)
PROD_DESC	NOT NULL	VARCHAR(25)
COST	NOT NULL	NUMBER(6,2)

下面是插入语句的范例：

```
INSERT INTO PRODUCTS_TBL  
VALUES ('7725', 'LEATHER GLOVES', 24.99);
```

1 row created.

在这个范例里，3个值被插入到一个具有3列的表里，值的顺序与列在表里的次序一致。前两个值使用了单引号包围，因为与之对应的列的数据类型为字符型。第3个值对应的列COST是数值型的，不需要使用单引号；当然，也可以使用单引号，而且不会对结果产生影响。

注意：引号的使用

数值型数据不必使用单引号，但其他数据类型都需要使用。换句话说，单引号对于数据库里的数值型数据来说是可选的，而对于其他数据类型来说是必需的。作为一种习惯，大多数SQL用户对数值型数据不使用单引号，这样可以提高查询命令的可读性。

### [5.2.2 给表里指定列插入数据](#)

有一种方法可以把数据插入到指定的列。举例来说，我们想插入除寻呼机号码之外的所有与雇员相关的数据，这时就必须在INSERT命令指定字段列表与值列表，如下所示：

```

INSERT INTO EMPLOYEE_TBL
(EMP_ID, LAST_NAME, FIRST_NAME, MIDDLE_NAME, ADDRESS, CITY, STATE, ZIP,
PHONE)
VALUES
('123456789', 'SMITH', 'JOHN', 'JAY', '12 BEACON CT',
'INDIANAPOLIS', 'IN', '46222', '3172996868');

1 row created.

```

给表中特定列插入数据的语法如下所示：

```

INSERT INTO TABLE_NAME ('COLUMN1', 'COLUMN2')
VALUES ('VALUE1', 'VALUE2');

```

在下面的范例里，我们向表ORDER\_TBL里的某些列插入数据。  
表的结构如下所示：

ORDERS\_TBL

COLUMN NAME	Null?	DATA TYPE
ORD_NUM	NOT NULL	VARCHAR2(10)
CUST_ID	NOT NULL	VARCHAR2(10)
PROD_ID	NOT NULL	VARCHAR2(10)
QTY	NOT NULL	NUMBER(4)
ORD_DATE		NULL DATE

INSERT的范例语句如下：

```

insert into orders_tbl (ord_num,cust_id,prod_id,qty)
values ('23A16','109','7725',2);

1 row created.

```

在INSERT语句里的表名称之后，我们在一对圆括号里指定了要插入数据的字段列表，其中只是没有包含ORD\_DATE。通过查看表定义可以看出，表里每条记录的ORD\_DATE字段都需要有值，这是因为它

没有被设置为NOT NULL，说明它可以是空的。注意，插入值的次序要与字段列表的次序相同。

注意：字段列表次序可以有差异

INSERT语句里的字段列表次序并不一定要与表定义中的字段次序相同，但插入值的次序要与字段列表的次序相同。除此之外，可以不用为列指定NULL，因为大部分RDBMS在默认情况下，允许列中出现NULL值。

### [5.2.3 从另一个表插入数据](#)

利用INSERT语句和SELECT语句的组合，我们可以根据对另一个表的查询结果把数据插入到表里。简单来说，查询是对数据库的一个质询，希望返回或不返回某些数据。关于查询的详细介绍请见第7章。查询就是用户向数据库提出的一个问题，而返回的数据就是答案。通过组合使用INSERT和SELECT语句，我们可以把从一个表的查询结果插入到另一个表里。

从另一个表插入数据的语法如下所示：

```
insert into table_name [('column1', 'column2')]  
select [*|('column1', 'column2')]  
from table_name  
[where condition(s)];
```

这里有3个新的关键字，分别是SELECT、FROM和WHERE，在此做一简要介绍。SELECT是SQL里执行查询的主要命令；FROM是查询中的一个子句，用于指定要进行查询的表的名称；WHERE子句也是查询的一部分，用于设置查询的条件。条件用于设置一个标准，从而决定哪些数据会受到影响，比如：WHERE NAME = 'SMITH'。这3个关键字将在第7章和第8章详细介绍。

下面的范例使用一个简单的查询来查看表 PRODUCTS\_TBL里的全

部数据。SELECT \*告诉数据库服务程序返回表里所有字段的数据。在此没有使用WHERE子句，所以会得到表里的全部记录。

```
select * from products_tbl;
PROD_ID  PROD_DESC                                COST
-----
11235    WITCH COSTUME                            29.99
222      PLASTIC PUMPKIN 18 INCH                  7.75
13       FALSE PARAFFIN TEETH                    1.1
90       LIGHTED LANTERNS                        14.5
15       ASSORTED COSTUMES                        10
9        CANDY CORN                          1.35
6        PUMPKIN CANDY                          1.45
87       PLASTIC SPIDERS                      1.05
119      ASSORTED MASKS                        4.95
1234     KEY CHAIN                            5.95
2345     OAK BOOKSHELF                         59.99

11 rows selected.
```

现在基于上述查询向表PRODUCTS\_TMP里插入数据，可以看到这个表里创建了11条记录。

```
insert into products_tmp
select * from products_tbl;

11 rows created.
```

在采用这种语法时，必须确保查询返回的字段与表里的字段或INSERT语句里指定的字段列表具有相同的次序。另外，还要确定SELECT语句返回的数据与要插入数据的表的字段具有兼容的数据类型。举例来说，如果想把一个值为'ABC'的 VARCHAR 字段插入到一个数值字段，就会导致语句失败。

下面的查询显示出表PRODUCTS\_TMP里刚刚插入的数据：



```
select * from products_tmp;
PROD_ID    PROD_DESC                                COST
-----
11235      WITCH COSTUME                             29.99
222        PLASTIC PUMPKIN 18 INCH                   7.75
13         FALSE PARAFFIN TEETH                     1.1
90         LIGHTED LANTERNS                      14.5
15         ASSORTED COSTUMES                       10
9          CANDY CORN                         1.35
6          PUMPKIN CANDY                        1.45
87         PLASTIC SPIDERS                     1.05
119        ASSORTED MASKS                      4.95
1234       KEY CHAIN                          5.95
2345       OAK BOOKSHELF                      59.99
```

11 rows selected.

#### [5.2.4 插入NULL值](#)

向表里的字段插入NULL值是相当简单的。当某一列的值不能确定时，我们可能需要向它插入一个NULL值。举例来说，不是每个人都有寻呼机，输入一个错误寻呼号码是不正确的，更何况这样会浪费存储空间。利用关键字NULL可以在列里插入NULL值。

插入NULL值的语法如下所示：

```
insert into schema.table_name values
('column1', NULL, 'column3');
```

关键字NULL应该位于正确的次序上，相应的字段会没有值的。在上面这个范例里，第二列的值会是NULL。

看一看下面这个范例：

```
insert into orders_tbl (ord_num,cust_id,prod_id,qty,ORD_DATE)
values ('23A16','109','7725',2,NULL);
```

1 row created.

在这个范例里，所有要插入数据的字段都列出来了，这也恰好是表 ORDERS\_TBL 里的全部字段。在此，NULL 值被插入到 ORD\_DATE 字段，表示或者不知道订购日期，或者目前还没有被订购。再来看一个范例：

```
insert into orders_tbl  
values ('23A16','109','7725',2);  
  
1 row created.
```

这条语句与前一条语句有两处不同，但结果是一样的。首先，这里没有字段列表。在向全部字段插入数据时，不必使用字段列表。其次，没有向 ORD\_DATE 字段插入 NULL 值，而是根本没有给它赋值，这意味着应该添加一个 NULL 值。记住，NULL 值表示字段没有值，与空字符串是不同的。

最后，考虑这样一种情况，PRODUCTS\_TBL 表中保存有 NULL 值，而用户需要将该表中的值插入 PRODUCTS\_TMP 表中，范例如下：

```

select * from products_tb;1
PROD_ID    PROD_DESC                                COST
-----
11235      WITCH COSTUME                                29.99
222        PLASTIC PUMPKIN 18 INCH                     7.75
13         FALSE PARAFFIN TEETH                      1.1
90         LIGHTED LANTERNS                       14.5
15         ASSORTED COSTUMES                        10
9          CANDY CORN                         1.35
6          PUMPKIN CANDY                          1.45
87         PLASTIC SPIDERS                     1.05
119        ASSORTED MASKS                      4.95
1234       NULL                               5.95
2345       OAK BOOKSHELF                      59.99
11 rows selected.

insert into products_tmp
select * from products_tbl;

11 rows created.

```

在这个范例中，如果数据即将插入的列允许接受NULL值，那么NULL值就可以直接插入该列。在后续章节中，我们会介绍如何为列指定默认值，这样，如果有NULL值被插入，就可以被自动转换成默认值。

## 5.3 更新现有数据

利用UPDATE命令可以修改表里的现有数据。这个命令不向表里添加新记录，也不删除记录，它只是修改现有的数据。它一般每次只更新数据库里的一个表，但可以同时更新表里的多个字段。根据需要，我们可以只更新表里的一行数据，也可以用一条语句就更新很多行数据。

### 5.3.1 更新一列的数据

UPDATE语句最简单的形式是用于更新表里的一列数据。在更新一列数据时，被更新的记录可以是一条，也可以是很多条。

更新一列的语法如下所示：

```
update table_name  
set column_name = 'value'  
[where condition];
```

下面的范例把表ORDERS\_TBL里ORD\_NUM值为'23A16'的记录（用WHERE子句指定）的QTY字段更新为1。

```
update orders_tbl  
set qty = 1  
where ord_num = '23A16';  
  
1 row updated.
```

下面这个范例与前面相同，只是没有了WHERE子句：

```
update orders_tbl  
set qty = 1;  
  
11 rows updated.
```

注意到在这个范例里，11条记录被更新了。这个语句把字段QTY设置为1，更新了表ORDERS\_TBL里的全部记录。这是我们想要的结果吗？有时是的，但一般我们很少使用没有WHERE子句的UPDATE语句。检查目标数据集是否正确的一种简单方式是对同一个表使用SELECT语句，其中包含要在UPDATE语句里使用的WHERE子句，判断返回的结果是否是我们要更新的记录。

**警告：**小心使用**UPDATE**和**DELETE**命令

在使用没有WHERE子句的UPDATE命令时要特别小心。如果没有使用WHERE子句设置条件，表里所有记录的相应字段都会被更新。在大多数情况下，DML命令都需要使用WHERE子句。

### 5.3.2 更新一条或多记录里的多个字段

下面来介绍如何使用一条UPDATE语句更新多个字段，其语法如下所示：

```
update table_name
set column1 = 'value',
    [column2 = 'value',]
    [column3 = 'value']
[where condition];
```

注意其中使用的SET——这里只有一个SET，但是有多个列，每个列之间以逗号分隔。可以看出SQL里的一种趋势：通常使用逗号来分隔不同类型的参数。下面的代码使用逗号分隔要更新的两列。同样，WHERE子句是可选的，但通常是必要的。

```
update orders_tbl
set qty = 1,
    cust_id = '221'
where ord_num = '23A16';

1 row updated.
```

注意：如何使用**SET**关键字

在每个UPDATE语句里，关键字SET只能使用一次。如果需要一次更新多个字段，就要使用逗号来分隔这些字段。

本书的后续章节将介绍如何使用更复杂的命令，利用一个或多个外部表来更新当前表中的字段，这需要使用JOIN命令。

## 5.4 从表里删除数据

DELETE命令用于从表里删除整行数据。它不能删除某一列的数据，而是删除行里全部字段的数据。使用DELETE语句一定要谨慎，因

为它一向很有效。

警告：不要省略**WHERE**子句

如果**DELETE**语句里没有**WHERE**子句，表里的所有数据都会被删除。作为一条规则，**DELETE** 语句应该总是使用 **WHERE** 子句。另外，还应该首先使用**SELECT**语句对**WHERE**子句进行测试。

另外，**DELETE**语句可能对数据库造成永久的影响。理想状态下，利用备份可以恢复被误删除的数据，但在有些情况下，这可能是很困难的，甚至是不可能的。如果数据不能被恢复，就只能重新输入到数据库。如果只是一行数据，这还算不上什么，但对于数以千记的记录来说，可不是什么小事。这就是**WHERE**子句的重要性。

使用下面的语法从表里删除一行或多行：

```
delete from table_name
[where condition];

delete from orders_tbl
where ord_num = '23A16';

1 row deleted.
```

注意**WHERE**子句的使用。如果要从表里删除指定的数据行，**WHERE**子句是必须的。我们几乎不会使用没有**WHERE**子句的**DELETE**语句，如果非要这么做，其结果类似如下范例：

```
delete from orders_tbl;

11 rows deleted.
```

前面范例里根据原始表数据进行填充而得到了一个临时表，在把**DELETE** 或 **UPDATE**语句应用于原始表之前，用临时表进行测试是一个很好的方法。可以使用我们在学习**UPDATE**语句时掌握的各种技巧。

在删除数据前，可以先使用**SELECT**语句对**DELETE**语句的**WHERE**子句进行测试。这样做可以对即将删除的数据进行验证，以确保操作无误。

## 5.5 小结

本章介绍了DML里的3个基本命令：**INSERT**、**UPDATE**和**DELETE**。显然，数据操作是SQL的一种强大功能，让用户能够用新数据填充表、更新现有数据和删除数据。

在对数据库里的数据进行更新或删除操作时，有时忽略了**WHERE**子句会让我们得到深刻的教训。在需要对特定记录进行操作时，特别是进行更新和删除操作时，一定要使用**WHERE**子句在SQL语句里设置条件，否则就会影响到目标表里的全部数据，这对数据库来说可能是一场灾难。在进行数据操作时，要注意保护数据并保持谨慎。

## 5.6 问与答

问：在看到这么多关于**DELETE**和**UPDATE**命令的警告之后，我有点不敢使用它们了。如果由于没有使用**WHERE**子句而意外地更新了表里的全部数据，怎样才能恢复它们呢？

答：没有必要担心，对数据库的操作大多数都可以被恢复，虽然可能需要相当的时间和工作。第6章将介绍事务控制的概念，它可以认可或撤销数据操作行为。

问：**INSERT**语句是向表里插入数据的唯一方式吗？

答：不，但**INSERT**语句是ANSI标准。各种实现都具有自己的工具以便向表里输入数据，比如Oracle有个名为SQL\*Loader的工具，很多实现都有名为**IMPORT**的工具用来插入数据。市面上有很多关于这些工具的详细介绍。

## 5.7 实践

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习是为了把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### 5.7.1 测验

使用具有如下结构的表EMPLOYEE\_TBL:

Column	data type	(not)null	
last_name	varchar2(20)	not null	
first_name	varchar2(20)	not null	
ssn	char(9)	not null	
phone	number(10)	null	
LAST_NAME	FIRST_NAME	SSN	PHONE
SMITH	JOHN	312456788	3174549923
ROBERTS	LISA	232118857	3175452321
SMITH	SUE	443221989	3178398712
PIERCE	BILLY	310239856	3176763990

下列语句运行后会有什么结果？

a.

```
insert into employee_tbl  
( 'JACKSON', 'STEVE', '313546078', '3178523443');
```

b.

```
insert into employee_tbl values  
( 'JACKSON', 'STEVE', '313546078', '3178523443');
```

c.



```
insert into employee_tbl values  
( 'MILLER', 'DANIEL', '230980012', NULL);
```

**d.**

```
insert into employee_tbl values  
( 'TAYLOR', NULL, '445761212', '3179221331');
```

**e.**

```
delete from employee_tbl;
```

**f.**

```
delete from employee_tbl  
where last_name = 'SMITH';
```

**g.**

```
delete from employee_tbl  
where last_name = 'SMITH'  
and first_name = 'JOHN';
```

**h.**

```
update employee_tbl  
set last_name = 'CONRAD';
```

**i.**

```
update employee_tbl  
set last_name = 'CONRAD'  
where last_name = 'SMITH';
```

**j**

```
update employee_tbl
set last_name = 'CONRAD',
first_name = 'LARRY';
```

k.

```
update employee_tbl
set last_name = 'CONRAD'
first_name = 'LARRY'
where ssn = '313546078';
```

### 5.7.2 练习

1. 到附录E，然后运行你的RDBMS。

现在需要向第3章练习创建的表里插入数据。小心地输入并执行附录E中的语句，向表中插入数据。输入完成后，就可以进行后续的练习了。

2. 使用表PRODUCTS\_TBL进行下面的练习。

向产品表添加如下产品：

PROD_ID	PROD_DESC	COST
301	FIREMAN COSTUME	24.99
302	POLICEMAN COSTUME	24.99
303	KIDDIE GRAB BAG	4.99

利用DML修改所添加的两种服装的价格，它们应该与witch costume同价。

现在要缩减产品线，首先对新产品下手。删除刚刚添加的3种产品。

在执行DELETE语句之前，有什么办法可以用来确定所删除的数据准确无误呢？

## 第6章 管理数据库事务

本章的重点包括：

事务的定义

用于控制事务的命令

事务命令的语法和范例

何时使用事务命令

低劣事务控制的后果

这一章将介绍数据库事务管理的概念。

## [6.1 什么是事务](#)

事务是对数据库执行的一个操作单位。它是以逻辑顺序完成的工作单元或工作序列，无论是用户手工操作，还是由程序进行的自动操作。在使用SQL的关系型数据库里，事务是由第5章介绍的数据操作语言（DML）完成的。事务是对数据库所做的一个或多个修改，比如利用UPDATE语句对表里某个人的姓名进行修改时，就是在执行一个事务。

一个事务可以是一个或多个DML语句。在管理事务时，任何指定的事务（DML语句组）都必须作为一个整体来完成，否则其中任何一条语句都不会完成。

下面是事务的本质特征：

所有的事务都有开始和结束；

事务可以被保存或撤销；

如果事务在中途失败，事务中的任何部分都不会被记录到数据库。

## [6.2 控制事务](#)

事务控制是对关系型数据库管理系统（RDBMS）里可能发生的各种事务的管理能力。在谈及事务时，我们是指前一章所介绍的INSERT、UPDATE和DELETE命令。

注意：事务的启动或执行在各个实现中是不同的，详细情况请查看具体实现的文档。

当一个事务被执行并成功完成时，虽然从输出结果来看目标表已经被修改了，但实际上目标表并不是立即被修改。当事务成功完成时，利用事务控制命令最终认可这个事务，可以把事务所做的修改保存到数据库，也可以撤销事务所做的修改。

控制事务的命令有3个：

**COMMIT;**

**ROLLBACK;**

**SAVEPOINT。**

下面的小节将详细介绍这3个命令。

注意：什么时候可以使用事务

事务控制命令只与DML命令INSERT、UPDATE和DELETE配合使用，比如我们不会在创建表之后使用COMMIT语句，因为当表被创建之后，它会自动被提交给数据库。也不能使用 ROLLBACK 语句来恢复被撤销的表。此外，还有其他类似的语句，也是不能被撤销的，例如 TRUNCATE语句。所以，在运行新的命令前，最好先确认一下用户所使用的RDBMS在事务方面的相关规定。当事务完成之后，事务信息被保存在数据库里的指定区域或临时回退区域。所有的修改都被保存到这个临时回退区域，直到事务控制命令出现。当事务控制命令出现时，所做的修改要么被保存到数据库，要么被放弃，然后临时回退区域被清空。图6.1展示了修改操作如何应用到关系型数据库。

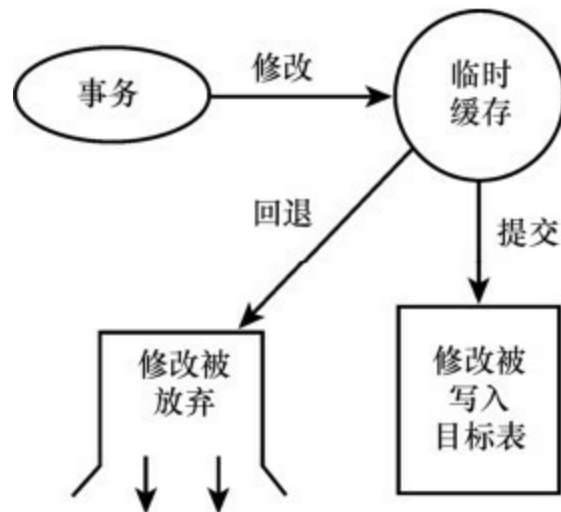


图6.1 回退区域

### 6.2.1 COMMIT命令

COMMIT命令用于把事务所做的修改保存到数据库，它把上一个COMMIT或ROLLBACK命令之后的全部事务都保存到数据库。

这个命令的语法是：

```
commit [ work ];
```

关键字COMMIT是语法中唯一不可缺少的部分，其后是用于终止语句的字符或命令，具体内容取决于不同的实现。关键字WORK是个选项，其唯一作用是让命令对用户更加友好。

在下面这个范例里，我们首先从查询表PRODUCT\_TMP里的全部数据开始：

```
SELECT * FROM PRODUCTS_TMP;
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95
2345	OAK BOOKSHELF	59.99

```
11 rows selected.
```

接下来，删除表里所有价格低于\$14.00的产品。

```
DELETE FROM PRODUCTS_TMP  
WHERE COST < 14;
```

```
8 rows deleted.
```

使用一个COMMIT语句把修改保存到数据库，完成这个事务。

```
COMMIT;
```

```
Commit complete.
```

对于数据库的大规模数据加载或撤销来说，应该多使用 COMMIT 语句；然而，过多的COMMIT语句会让工作需要大量额外时间才能完成。记住，全部修改都首先被送到临时回退区域，如果这个临时回退区域没有空间了，不能保存对数据库所做的修改，数据库很可能会挂起，禁止进一步的事务操作。

实际上，在提交了一条UPDATE、INSERT或DELETE语句之后，大部分RDBMS都是使用事务来进行后台处理的，一旦操作被取消或报错，所做的操作就可以被撤销。所以，在提交了一个事务之后，会有一些操作来确保事务正常运行。在现实生活中，用户可能会在ATM上提交一个银行事务以便从自己的账户中取出现金。这时，就需要完成取钱和更新账户余额两项事务。很显然，我们希望这两项事务能够同时完成，或者全部失败。否则，系统数据的完整性就会受到影响。所以，在这个实例中，我们会将两项操作合并为一个事务，来确保对操作结果的控制。

注意：不同的实现对**COMMIT**命令的提交有所不同

在某些实现里，事务不是通过使用COMMIT命令提交的，而是由退出数据库的操作引发提交。但是在其他实现里，比如MySQL，在执行SET TRANSACTION命令之后，在数据库收到COMMIT或ROLLBACK之前，自动提交功能是不会恢复的。此外，在Microsoft SQL Server中，除非事务正在运行，否则语句会被自动提交。所以，用户务必要了解所使用的RDBMS在事务处理和命令提交方面的相关规定。

### 6.2.2 ROLLBACK命令

ROLLBACK 命令用于撤销还没有被保存到数据库的命令，它只能用于撤销上一个COMMIT或ROLLBACK命令之后的事务。

ROLLBACK的语法如下所示：

```
rollback [ work ];
```

与COMMIT命令一样的是，关键字WORK只是个选项。

在下面的范例里，首先选择表PRODUCTS\_TMP里的全部记录，这是前一次删除14条记录之后所剩的数据。

```
SELECT * FROM PRODUCTS_TMP;
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
90	LIGHTED LANTERNS	14.5
2345	OAK BOOKSHELF	59.99

```
3 rows selected.
```

接下来更新表，把标识为11235的产品价格修改为\$39.99:

```
update products_tmp  
set cost = 39.99  
where prod_id = '11235';
```

```
1 row updated.
```

现在对表进行一个简单的查询，可以发现修改似乎已经生效了:

```
select * from products_tmp;
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	39.99
90	LIGHTED LANTERNS	14.5
2345	OAK BOOKSHELF	59.99

```
3 rows selected.
```

现在，执行ROLLBACK命令来撤销刚刚所做的修改:

```
rollback;
```

```
Rollback complete.
```

最后，验证所做的修改并没有被提交到数据库:



```
select * from products_tmp;
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
90	LIGHTED LANTERNS	14.5
2345	OAK BOOKSHELF	59.99

```
3 rows selected
```

### [6.2.3 SAVEPOINT命令](#)

保存点是事务过程中的一个逻辑点，我们可以把事务回退到这个点，而不必回退整个事务。

SAVEPOINT命令的语法如下：

```
savepoint savepoint_name
```

这个命令就是在事务语句之间创建一个保存点。ROLLBACK 命令可以撤销一组事务操作，而保存点可以将大量事务操作划分为较小的、更易于管理的组。

Microsoft SQL Server的语法稍有不同。在SQL Server中，使用的是SAVE TRANSACTION，而不是SAVEPOINT，范例如下：

```
save transaction savepoint_name
```

除此之外，SQL Server与其他数据库实现完全相同。

### [6.2.4 ROLLBACK TO SAVEPOINT命令](#)

回退到保存点的命令语法如下：

```
ROLLBACK TO SAVEPOINT_NAME;
```

在下面的范例里，我们要从表PRODUCTS\_TMP表里删除剩余的数

据，在进行每次删除之前都使用SAVEPOINT命令，这样就可以在任何时候利用ROLLBACK命令回退到任意一个保存点，从而把适当的数据恢复到原始状态：

```
savepoint sp1;

Savepoint created.

delete from products_tmp where prod_id = '11235';

1 row deleted.

savepoint sp2;

Savepoint created.

delete from products_tmp where prod_id = '90';

1 row deleted.

savepoint sp3;

Savepoint created.

delete from products_tmp where prod_id = '2345';

1 row deleted.
```

注意：保存点的名称必须唯一

在相应的事务操作组里，保存点的名称必须是唯一的，但其名称可以与表或其他对象的名称相同，详细的命名规范请见具体实现的说明文档。保存点名称的设置属于个人喜好，它只被数据库开发人员用来管理事务操作组。

在三次删除操作完成之后，假设我们又改变了主意，决定回退到名为SP2的保存点。由于SP2是在第一次删除操作之后创建的，所以这样

做会撤销最后两次删除操作：

```
rollback to sp2;
```

```
Rollback complete.
```

现在查看表里的内容，可以发现只发生了第一次删除操作：

```
select * from products_tmp;
```

PROD_ID	PROD_DESC	COST
90	LIGHTED LANTERNS	14.5
2345	OAK BOOKSHELF	59.99

```
2 rows selected.
```

记住，ROLLBACK命令本身会回退到上一个COMMIT或ROLLBACK语句。由于我们还没有执行COMMIT命令，所以这时执行ROLLBACK命令会撤销全部删除命令，如下所示：

```
rollback;
```

```
Rollback complete.
```

```
select * from products_tmp;
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
90	LIGHTED LANTERNS	14.5
2345	OAK BOOKSHELF	59.99

```
3 rows selected.
```

### [\*\*6.2.5 RELEASE SAVEPOINT命令\*\*](#)

这个命令用于删除创建的保存点。在某个保存点被释放之后，就不能再利用ROLLBACK命令来撤销这个保存点之后的事务操作了。利用这个命令可以避免意外地回退到某个不再需要的保存点。

```
RELEASE SAVEPOINT savepoint_name;
```

Microsoft SQL Server不支持RELEASE SAVEPOINT命令；在事务完成以后，所有的保存点会被自动删除。这个过程不必使用COMMIT或者ROLLBACK命令。用户在自己的环境中创建事务时，需要牢记这一点。

#### **6.2.6 SET TRANSACTION命令**

这个命令用于初始化数据库事务，可以指定事务的特性。举例来说，我们可以指定事务是只读的或是可以读写的，如下所示：

```
SET TRANSACTION READ WRITE;  
SET TRANSACTION READ ONLY;
```

READ WRITE用于对数据库进行查询和操作数据的事务，READ ONLY用于只进行查询的事务。READ ONLY很适合生成报告，而且能够提高事务完成的速度。如果事务是READ WRITE类型的，数据库必须对数据库对象进行加锁，从而在多个事务同时发生时保持数据完整性。如果事务是READ ONLY，数据库就不会创建锁定，这样就会提高事务的性能。

事务还可以设置其他特性，但超出了本书的讨论范围。MySQL通过对事务实现不同级别的隔离来实现类似功能，但语法略有不同。详细情况请参考具体实现的帮助文档。

### **6.3 事务控制与数据库性能**

低劣的事务控制会降低数据库性能，甚至导致数据库异常终止。反复出现的数据库性能恶化可能是由于在大量插入、更新或删除中缺少事务控制。大规模批处理还会导致临时存储的回退信息不断膨胀，直到出现COMMIT或ROLLBACK命令。

当出现 COMMIT 命令时，回退事务信息被写入到目标表里，临时存储区域里的回退信息被清除。当出现ROLLBACK命令时，修改不会作用于数据库，而临时存储区域里的回退信息被清除。如果一直没有出现COMMIT或ROLLBACK命令，临时存储区域里的回退信息就会不断增长，直到没有剩余空间，导致数据库停止全部进程，直到空间被释放。虽然存储空间的使用实际上是由数据库管理员（DBA）控制的，但缺少事务控制还是会导致数据库处理停止，有时迫使DBA采取的行动会中止正在运行的用户进程。

## [6.4 小结](#)

这一章通过介绍3个事务控制命令（COMMIT、ROLLBACK和SAVEPOINT）展示了事务管理的初步概念。COMMIT用于把事务保存到数据库，ROLLBACK用于撤销已经执行的事务，而SAVEPOINT用于把事务划分为组，让我们可以回退到事务过程中特定的逻辑位置。

在运行大规模事务操作时，应该经常使用 COMMIT 和 ROLLBACK 命令来保证数据库具有足够的剩余空间。另外还要记住，这些事务命令只用于3个DML命令：INSERT、UPDATE和DELETE。

## [6.5 问与答](#)

问：每个INSERT语句是否都需要执行一个COMMIT？

答：不，绝对不需要。如果要向表里插入几十万条记录，建议每 5 000~10 000条记录执行一个COMMIT语句，具体数值取决于临时回退区

域的大小（向数据库管理员寻求建议）。当回退区域没有空间时，数据库可能停止或工作不正常。

问：**ROLLBACK**命令如何撤销一个事务？

答：**ROLLBACK**命令清除回退区域里的全部修改。

问：在执行事务过程中，如果**99%**的事务都完成了，但另外**1%**出现了错误，能否只重做出现错误的部分呢？

答：不能，整个事务必须是成功的，否则数据完整性就会遭到破坏。

问：在执行**COMMIT**语句之后，事务操作的效果就是永久的了，但使用**UPDATE**命令不是能够修改数据吗？

答：“永久”一词在此是表示它现在是数据库的一部分了。**UPDATE**语句当然一直都可以用于修改数据。

## **6.6 实践**

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习是为了把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### **6.6.1 测验**

1. 判断正误：如果提交了一些事务，还有一些事务没有提交，这时执行**ROLLBACK**命令，同一过程里的全部事务都会被撤销。
2. 判断正误：**SAVEPOINT**命令会把一定数量已执行事务之后的事务保存起来。
3. 简要叙述下面每个命令的作用：**COMMIT**、**ROLLBACK**和**SAVEPOINT**。
4. 在Microsoft SQL Server中执行事务有什么不同点？

## 5. 使用事务进行操作的实质是什么？

### 6.6.2 练习

1. 执行如下事务，并且在第 3 个事务之后创建一个保存点或者一个保存事务，然后在最后执行一条ROLLBACK命令。请说明上述操作完成之后，表CUSTOMER\_TBL的内容。

```
INSERT INTO CUSTOMER_TBL VALUES(615,'FRED WOLF','109 MEMORY  
LANE','PLAINFIELD','IN',46113,'3175555555',NULL);  
INSERT INTO CUSTOMER_TBL VALUES(559,'RITA THOMPSON','125  
PEACHTREE','INDIANAPOLIS','IN',46248,'3171111111',NULL);  
INSERT INTO CUSTOMER_TBL VALUES(715,'BOB DIGGLER','1102 HUNTINGTON  
ST','SHELBY','IN',41234,'3172222222',NULL);  
UPDATE CUSTOMER_TBL SET CUST_NAME='FRED WOLF' WHERE CUST_ID='559';  
UPDATE CUSTOMER_TBL SET CUST_ADDRESS='APT C 4556 WATERWAY' WHERE  
CUST_ID='615';  
UPDATE CUSTOMER_TBL SET CUST_CITY='CHICAGO' WHERE CUST_ID='715';
```

2. 执行如下事务，在第3个事务之后创建一个保存点。

事务执行完之后添加一条COMMIT命令，之后再加上一条回退到保存点的ROLLBACK命令，这时会发生什么呢？

```
UPDATE CUSTOMER_TBL SET CUST_NAME='FRED WOLF' WHERE CUST_ID='559';  
UPDATE CUSTOMER_TBL SET CUST_ADDRESS='APT C 4556 WATERWAY' WHERE  
CUST_ID='615';  
UPDATE CUSTOMER_TBL SET CUST_CITY='CHICAGO' WHERE CUST_ID='715';  
DELETE FROM CUSTOMER_TBL WHERE CUST_ID='615';  
DELETE FROM CUSTOMER_TBL WHERE CUST_ID='559';  
DELETE FROM CUSTOMER_TBL WHERE CUST_ID='615';
```



## [第三部分 从查询中获得有效的结果](#)

第7章 数据库查询

第8章 使用操作符对数据进行分类

第9章 汇总查询得到的数据

第10章 数据排序与分组

第11章 调整数据的外观

第12章 日期和时间

### [第7章 数据库查询](#)

本章的重点包括：

什么是数据库查询

如何使用SELECT语句

利用WHERE子句为查询添加条件

使用列别名

从其他用户的表里选择数据

本章将介绍数据库查询，主要是SELECT语句的使用。在数据库创建之后，SQL命令里最常用的语句就是SELECT，它让我们可以查看数据库里保存的数据。

#### [7.1 什么是查询](#)

查询是使用SELECT语句对数据库进行探究。我们利用查询，根据需要从数据库里以一种可理解的格式提取数据。举例来说，假设我们有一个雇员表，就可能利用SQL语句返回哪个雇员得到最高的薪水。这种获取有用信息的请求是关系型数据库里典型的查询操作。



## 7.2 SELECT语句

SELECT 语句代表了 SQL 里的数据查询语言（DQL），是构成数据库查询的基本语句。它并不是一个单独的语句，也就是说，为了构成一个在句法上正确的查询，需要一个或多个条件子句（元素）。除了必要的子句，还有其他一些可选的子句可以增强SELECT语句的整体功能。SELECT语句绝对是SQL里功能最强大的。FROM子句是一条必要的子句，必须总是与SELECT联合使用。

SELECT语句里有4个关键字（或称为子句）是最有价值的，如下所示：

SELECT  
FROM  
WHERE  
ORDER BY

下面的小节将详细介绍这些关键字。

### 7.2.1 SELECT语句

SELECT语句与FROM子句联合使用，以一种有组织的、可读的方式从数据库提取数据。查询中的SELECT部分用于指定需要表里哪些字段的数据。

简单的SELECT语句的语法如下所示：

```
SELECT [ * | ALL | DISTINCT COLUMN1, COLUMN2 ]  
FROM TABLE1 [ , TABLE2 ];
```

在查询里，关键字 SELECT 后面是字段列表，它们是查询输出的组成部分。星号（\*）表示输出结果里包含表里的全部字段，其详细使用方式请查看相应实现的文档。选项ALL用于显示一列的全部值，包括重

复值。选项DISTINCT禁止在输出结果里包含重复的行。选项ALL是默认的操作方式，这意味着它并不在SELECT语句中明确指定。关键字FROM后面是一个或多个表的名称，用于指定数据的来源。SELECT语句后面的字段列表中使用逗号进行分隔，FROM子句里的表也是如此。

注意：使用逗号来分隔参数

在SQL语句的列表里，使用逗号分隔各个参数。参数是SQL语句或命令里必需的或可选的值。常见的参数列表包括查询中的字段列表、查询中的表列表、插入到表里的数据列表、WHERE子句里的条件。

下面的范例将展示SELECT语句的基本功能。首先，对表PRODUCTS\_TBL进行一个简单的查询：

```
SELECT * FROM PRODUCTS_TBL;
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95
2345	OAK BOOKSHELF	59.99

```
11 rows selected.
```

星号表示表里的全部字段，也就是PROD\_ID、PROD\_DESC和COST。字段在输出结果中显示的次序与其在表里的次序相同。表里一共有 11 条记录，这是由反馈信息“11 rows selected”反映出来的。反馈信息的表示方式在不同实现里有所区别，比如有些查询的反馈信息是“11

rows affected”。星号是书写SQL查询的一种行之有效的便捷方法，但实际操作时，&nbsp;最好还是明确地指出所要返回的字段名称。

现在从另一个表 CANDY\_TBL 里选择数据，这个表与 PRODUCTS\_TBL 具有同样的结构。在关键字SELECT之后列出字段名称，从而只显示表里的一个字段：

```
SELECT PROD_DESC FROM CANDY_TBL;
```

```
PROD_DESC
-----
CANDY CORN
CANDY CORN
HERSHEYS KISS
SMARTIES

4 rows selected.
```

表CANDY\_TBL里有4条记录。下面的语句使用选项ALL，其结果会展示出ALL完全是多余的，它是默认选项，不需要明确指定。

```
SELECT ALL PROD_DESC
FROM CANDY_TBL;
PROD_DESC
-----
CANDY CORN
CANDY CORN
HERSHEYS KISS
SMARTIES

4 rows selected.
```

下面的语句使用了DISTINCT选项，从而在显示中去除了重复记录，所以这一次CANDY CORN只显示了一次。

```
SELECT DISTINCT PROD_DESC  
FROM CANDY_TBL;
```

```
PROD_DESC  
-----  
CANDY CORN  
HERSHEYS KISS  
SMARTIES  
  
3 rows selected.
```

我们还可以用圆括号把选项DISTINCT和ALL与相应的字段包围在一起。在SQL及其他很多语言里，我们都利用圆括号来提高代码的可读性。

```
SELECT DISTINCT(PROD_DESC)  
FROM CANDY_TBL;
```

```
PROD_DESC  
-----  
CANDY CORN  
HERSHEYS KISS  
SMARTIES  
  
3 rows selected.
```

### [7.2.2 FROM子句](#)

FROM子句必须与SELECT语句联合使用，它是任何查询的必要元素，其作用是告诉数据库从哪些表里获取所需的数据，它可以指定一个或多个表，但必须至少指定一个表。

FORM子句的语法如下所示：

```
from table1 [ , table2 ]
```

### [7.2.3 WHERE子句](#)

查询里的条件指定了要返回满足什么标准的信息。条件的值是 TRUE或FALSE，从而限制查询中获取的数据。WHERE子句用于给查询添加条件，从而去除用户不需要的数据。

WHERE子句里可以有多个条件，它们之间以操作符 AND或 OR 连接，详细介绍请见第8章，届时还会介绍其他一些条件操作符。本章只介绍包含一个条件的查询。

操作符是SQL里的字符或关键字，用于连接SQL语句里的元素。

WHERE子句的语法如下所示：

```
select [ all | * | distinct column1, column2 ]
from table1 [ , table2 ]
where [ condition1 | expression1 ]
[ and|OR condition2 | expression2 ]
```

下面是一个没有WHERE子句的简单SELECT语句：

```
SELECT *
FROM PRODUCTS_TBL;
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95
2345	OAK BOOKSHELF	59.99

11 rows selected.

现在对这个查询添加条件：

```
SELECT * FROM PRODUCTS_TBL
WHERE COST < 5;
```

PROD_ID	PROD_DESC	COST
13	FALSE PARAFFIN TEETH	1.1
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95

5 rows selected.

这里只显示了价格小于\$5的记录。

下面这个查询显示了产品标识为119的产品描述和价格：

```
SELECT PROD_DESC, COST
FROM PRODUCTS_TBL
WHERE PROD_ID = '119';
```

PROD_DESC	COST
ASSORTED MASKS	4.95

1 row selected.

#### [7.2.4 ORDER BY子句](#)

我们一般需要让输出以某种方式进行排序，为此可以使用ORDER BY子句，它能够以用户指定的列表格式对查询结果进行排列。ORDER BY子句的默认次序是升序，也就是说，如果对输出为字符的结果进行排序，就是A到Z的次序。反之，降序就是以Z到A的次序显示字符结果。对于数字值来说，升序是从1到9，降序是从9到1。

ORDER BY子句的语法是：

```

select [ all | * | distinct column1, column2 ]
from table1 [ , table2 ]
where [ condition1 | expression1 ]
[ and|OR condition2 | expression2 ]
ORDER BY column1|integer [ ASC|DESC ]

```

对前面某个范例语句进行扩展来体会如何使用ORDER BY子句。比如以升序（也就是字母顺序）对产品描述进行排序。注意其中使用的ASC选项，它可以在ORDER BY子句中的任意一个字段之后出现。

```

SELECT PROD_DESC, PROD_ID, COST
FROM PRODUCTS_TBL
WHERE COST < 20
ORDER BY PROD_DESC ASC;

```

PROD_DESC	PROD_ID	COST
ASSORTED COSTUMES	15	10
ASSORTED MASKS	119	4.95
CANDY CORN	9	1.35
FALSE PARAFFIN TEETH	13	1.1
LIGHTED LANTERNS	90	14.5
PLASTIC PUMPKIN 18 INCH	222	7.75
PLASTIC SPIDERS	87	1.05
PUMPKIN CANDY	6	1.45

8 rows selected.

注意：排序方式

SQL排序是基于字符的ASCII排序。数字0~9会按其字符值进行排序，并且位于字母A到Z之前。由于数字值在排序时是被当作字符处理的，所以下面这些数字的排序是这样的：1、12、2、255、3。

下面的范例语句里使用了DESC，把输出结果按照反字母顺序显示：

```
SELECT PROD_DESC, PROD_ID, COST
FROM PRODUCTS_TBL
WHERE COST < 20
ORDER BY PROD_DESC DESC;
```

PROD_DESC	PROD_ID	COST
PUMPKIN CANDY	6	1.45
PLASTIC SPIDERS	87	1.05
PLASTIC PUMPKIN 18 INCH	222	7.75
LIGHTED LANTERNS	90	14.5
FALSE PARAFFIN TEETH	13	1.1
CANDY CORN	9	1.35
ASSORTED MASKS	119	4.95
ASSORTED COSTUMES	15	10

8 rows selected.

注意：默认排序方式

由于升序是默认的排序方式，所以ASC选项并不需要明确指定。

SQL里存在着一些简化方式。ORDER BY子句里的字段可以缩写为一个整数，这个整数取代了实际的字段名称（排序操作中使用的一个别名），表示字段在关键字 SELECT 之后列表里的位置。

下面是在ORDER BY子句里使用整数表示字段的范例：



```
SELECT PROD_DESC, PROD_ID, COST
FROM PRODUCTS_TBL
WHERE COST < 20
ORDER BY 1;
```

PROD_DESC	PROD_ID	COST
ASSORTED COSTUMES	15	10
ASSORTED MASKS	119	4.95
CANDY CORN	9	1.35
FALSE PARAFFIN TEETH	13	1.1
LIGHTED LANTERNS	90	14.5
PLASTIC PUMPKIN 18 INCH	222	7.75
PLASTIC SPIDERS	87	1.05
PUMPKIN CANDY	6	1.45

8 rows selected.

在这个查询里，整数1代表字段PROD\_DESC，2代表PROD\_ID，而3代表COST，以此类推。

在一个查询里可以对多个字段进行排序，这时可以使用字段名或相应的整数：

```
ORDER BY 1,2,3
```

ORDER BY子句里的字段次序不一定要与关键字 SELECT之后的字段次序一致，如下所示：

```
ORDER BY 1,3,2
```

ORDER BY子句里指定的字段次序决定了排序过程的完成方式。下面这个语句将首先对字段PROD\_DESC进行排序，再对字段COST进行排序。

```
ORDER BY PROD_DESC,COST
```

### 7.2.5 大小写敏感性

在使用 SQL 编写代码时，大小写敏感性是一个需要理解的重要概念。一般来说，SQL 命令和关键字是不区分大小写的，也就是允许我们以大写或小写来输入命令和关键字，而且可以混用，大小写混用通常被称为驼峰命名法。关于大小写的问题请见第5章的介绍。

排序规则（collation）决定了 RDBMS 如何解释数据，包括排序方式和大小写敏感性等内容。数据的大小写敏感性很重要，这直接决定了 WHERE 子句如何匹配记录。用户务必要明确所用的 RDBMS 在排序规则方面的相关规定。在某些系统中，例如 MySQL 和 Microsoft SQL Server，默认是大小写不敏感的。这就意味着，在进行数据匹配时，系统会忽视数据的大小写。也有一些系统，例如 Oracle，默认是大小写敏感的。这种系统在进行数据匹配时，需要考虑大小写情况。大小写敏感性取决于所用的数据库，因此在不同的系统中对查询的影响就会相应地有所不同。

注意：使用标准的大小写形式

在从数据库里获取数据时，必须在查询里使用与数据一致的大小写。此外，最好能够实施公司级别的大小写规则，确保在公司内部以统一的方式处理数据输入。

然而，对于在用户的 RDBMS 中确保数据的一致性来讲，大小写就是一个需要考虑的问题。在大多数情况下，数据在关系型数据库里似乎都是以大写形式保存的，以便保持数据的一致性。

举例来说，如果使用随意的大小写方式输入数据，数据的一致性就可能被破坏：

```
SMITH  
Smith  
smith
```

如果某人的姓被存储为smith，而我们在Oracle等大小写敏感的RDBMS中执行了如下查询，就不会得到返回结果：

```
SELECT *  
FROM EMPLOYEE_TBL  
WHERE LAST_NAME = 'SMITH';  
  
SELECT *  
FROM EMPLOYEE_TBL  
WHERE UPPER(LAST_NAME) = UPPER('Smith');
```

### 7.3 简单查询的范例

下面的小节基于前面介绍的概念展示了查询的一些范例，首先是最简单的查询，然后逐步丰富它。在此我们使用表EMPLOYEE\_TBL。

从表里选择全部记录，显示全部字段：

```
SELECT * FROM EMPLOYEE_TBL;
```

从表里选择全部记录，显示指定的字段：

```
SELECT EMP_ID  
FROM EMPLOYEE_TBL;
```

注意：克服大小写敏感问题

在类似 Oracle 这样对大小写敏感的系统里，往往需要不断地核对数据，或者使用后续章节中介绍的SQL函数来修改数据，以便克服这类大小写问题。下面的范例演示了如何使用UPPER函数来改变WHERE子句所涉数据的大小写。

从表里选择全部记录，显示指定的字段。命令可以在一行输入，或是根据喜好使用软掉头：

```
SELECT EMP_ID FROM EMPLOYEE_TBL;
```

从表里选择全部记录，显示多个字段：

```
SELECT EMP_ID, LAST_NAME  
FROM EMPLOYEE_TBL;
```

显示满足指定条件的数据：

```
SELECT EMP_ID, LAST_NAME  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = '33333333';
```

注意：确保所做的查询有约束条件

在从一个庞大的表里返回全部记录时，会得到大量的数据。

显示满足指定条件的数据，对输出结果进行排序：

```
SELECT EMP_ID, LAST_NAME  
FROM EMPLOYEE_TBL  
  
WHERE CITY = 'INDIANAPOLIS'  
ORDER BY EMP_ID;
```

显示满足指定条件的数据，根据多个字段进行排序，其中一个字段是逆序。在下面的范例中，EMP\_ID以升序排列，而LAST\_NAME 则以降序排列：

```
SELECT EMP_ID, LAST_NAME  
FROM EMPLOYEE_TBL  
WHERE CITY = 'INDIANAPOLIS'  
ORDER BY EMP_ID, LAST_NAME DESC;
```

显示满足指定条件的数据，利用整数代替字段名来表示要排序的字段：

```
SELECT EMP_ID, LAST_NAME
FROM EMPLOYEE_TBL
WHERE CITY = 'INDIANAPOLIS'
ORDER BY 1;
```

显示满足指定条件的数据，利用整数指定要排序的多个字段。字段的排序次序与它们在SELECT之后的次序并不相同：

```
SELECT EMP_ID, LAST_NAME
FROM EMPLOYEE_TBL
WHERE CITY = 'INDIANAPOLIS'
ORDER BY 2, 1;
```

### [7.3.1 统计表里的记录数量](#)

利用一个简单的查询就可以了解表里的记录数量，或是某个字段里值的数量。统计工作是由函数COUNT完成的。虽然关于函数的内容要在本书稍后才有介绍，但在此引入这个函数是因为它经常出现在简单的查询之中。

COUNT函数的语法如下所示：

```
SELECT COUNT(*)
FROM TABLE_NAME;
```

COUNT函数使用一对圆括号来指定目标字段，或是一个星号表示统计表里的全部记录。

注意：基本统计

如果被统计的字段是NOT NULL（必填字段），那么其值的数量就与表里记录的数量相同。但一般来说，我们使用COUNT（\*）来统计表里的记录数量。

下面的语句可以统计表PRODUCTS\_TBL里的记录数量：

```
SELECT COUNT(*) FROM PRODUCTS_TBL;
```

```
COUNT(*)  
-----  
          9
```

```
1 row selected.
```

下面的语句统计表PRODUCTS\_TBL里字段PROD\_ID的值的数量：

```
SELECT COUNT(PROD_ID) FROM PRODUCTS_TBL;
```

```
COUNT(PROD_ID)  
-----  
          9
```

```
1 row selected.
```

如果要统计表中特定列所出现的值的种类数，需要在COUNT函数中使用DISTINCT关键字。例如，如果要统计EMPLOYEE\_TBL表的STATE列中不同值的种类数，需要使用如下查询：

```
SELECT COUNT(DISTINCT PROD_ID) FROM PRODUCTS_TBL;
```

```
COUNT(DISTINCT PROD_ID)  
-----  
          1
```

### [7.3.2 从另一个用户表里选择数据](#)

要访问另一个用户的表，必须拥有相应的权限，否则就不能进行访问。在获得准许之后，我们可以从其他用户的表里获取数据（GRANT命令将在第20章介绍）。为了在SELECT语句里访问另一个用户的表，必须在表的名称之前添加规划名或相应的用户名，如下所示：

```
SELECT EMP_ID
FROM SCHEMA.EMPLOYEE_TBL;
```

### 7.3.3 使用字段别名

在进行某些查询时，我们使用字段别名来临时命名表的字段，其语法如下所示：

```
SELECT COLUMN_NAME ALIAS_NAME
FROM TABLE_NAME;
```

下面的范例显示了产品描述两次，并且给第二个字段一个别名：PRODUCT。请注意输出的字段标题。

```
select prod_desc,
       prod_desc product
from products_tbl;
```

PROD_DESC	PRODUCT
WITCH COSTUME	WITCH COSTUME
PLASTIC PUMPKIN 18 INCH	PLASTIC PUMPKIN 18 INCH
FALSE PARAFFIN TEETH	FALSE PARAFFIN TEETH
LIGHTED LANTERNS	LIGHTED LANTERNS
ASSORTED COSTUMES	ASSORTED COSTUMES
CANDY CORN	CANDY CORN
PUMPKIN CANDY	PUMPKIN CANDY
PLASTIC SPIDERS	PLASTIC SPIDERS
ASSORTED MASKS	ASSORTED MASKS
KEY CHAIN	KEY CHAIN
OAK BOOKSHELF	OAK BOOKSHELF

```
11 rows selected.
```

注意：在查询中使用别名

如果要访问的表在数据库里有别名，就可以不必指定表的规划名。

别名就是表的另一个名称，详细讨论请见第21章。

利用字段别名可以自定义字段的标题，在某些SQL实现里，字段别名还可以让我们用比较简洁的名称来引用某个字段。

注意：在查询中重新命名字段

当字段名称在SELECT语句里被重新命名时，其名称实际上并没有被修改，这种改变只在特定的SELECT语句里有效。

## [7.4 小结](#)

本章简单介绍了数据库查询的概念，这是从关系型数据库获取有用数据的手段。SELECT语句是一种数据查询语言（DQL）命令，用于在SQL里创建查询。每个SELECT语句里都必须包含FROM子句。另外，利用WHERE子句可以为查询设置条件，利用ORDER BY子句可以为数据进行排序。本章介绍了编写查询语句的基础知识，后面的章节将进行更详细和深入的介绍。

## [7.5 问与答](#)

问：为什么SELECT子句没有FROM子句就不行？

答：SELECT子句只是告诉数据库我们需要什么样的数据，而FROM子句告诉数据库到什么地方来获取这些数据。

问：在使用ORDER BY子句并设置为降序排序时，对数据到底有什么影响呢？

答：假设我们使用了ORDER BY子句，并且从表EMPLOYEE\_TBL选择了字段last\_name。如果选择了降序排序，其次序就是从字母Z开始，到字母A结束。假设使用了ORDER BY子句，并且从表EMPLOYEE\_PAY\_TBL里选择的表示薪水的字段，这时选择降序排序就会从最高薪水开始，到最低薪水结束。



问：重新命名字段有什么好处？

答：新名称可以在特定报告中更好地描述所返回的数据。

问：下面语句的排序是什么？

```
SELECT PROD_DESC,PROD_ID,COST FROM PRODUCTS_TBL  
ORDER BY 3,1
```

答：查询会首先以COST字段进行排序，然后再以PROD\_DESC进行排序。由于没有指定排序方式，所以两者都会是默认的升序。

## [7.6 实践](#)

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习是为了把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### [7.6.1 测验](#)

1. 说出任何SELECT语句都需要的组成部分。
2. 在WHERE子句里，任何数据都需要使用单引号吗？
3. SELECT语句属于SQL语言里的哪一类命令？
4. WHERE子句里能使用多个条件吗？
5. DISTINCT选项的作用是什么？
6. 选项ALL是必需的吗？
7. 在基于字符字段进行排序时，数字字符是如何处理的？
8. 在大小写敏感性方面，Oracle与MySQL和Microsoft SQL Server有什么不同？

### [7.6.2 练习](#)

1. 在计算机上运行RDBMS。使用数据库learnsql，输入以下SELECT命令。判断其语法是否正确，如果不正确就进行必要的修改。这里使用的是表EMPLOYEE\_TBL。

a.

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME,  
FROM EMPLOYEE_TBL;
```

b.

```
SELECT EMP_ID, LAST_NAME  
ORDER BY EMPLOYEE_TBL  
FROM EMPLOYEE_TBL;
```

c.

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = '213764555'  
ORDER BY EMP_ID;
```

d.

```
SELECT EMP_ID SSN, LAST_NAME  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = '213764555'  
ORDER BY 1;
```

e.

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = '213764555'  
ORDER BY 3, 1, 2;
```

2. 下面这个SELECT语句能工作吗？

```
SELECT LAST_NAME, FIRST_NAME, PHONE  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = '33333333';
```

3. 编写一条SELECT语句，从表PRODUCTS\_TBL里返回每件产品的名称和价格。哪个产品是最贵的？
4. 编写一个查询，生成全部顾客及其电话号码的列表。
5. 编写一个查询，生成具有某个特定姓的顾客的列表。尝试在WHERE子句中，使用混合大小写和全部大写两种方式。确定用户使用的RDBMS是否为大小写敏感。

## [第8章 使用操作符对数据进行分类](#)

本章的重点包括：

什么是操作符

SQL里操作符的概述

操作符如何单独使用

操作符如何联合使用

操作符用于在SELECT命令的WHERE子句中为返回的数据指定更明确的条件。SQL里有多种操作符，可以满足各种不同的查询需要。本章将介绍操作符的种类，以及如何在WHERE子句中正确使用操作符。

### [8.1 什么是SQL里的操作符](#)

操作符是一个保留字或字符，主要用于SQL语句的WHERE子句来执行操作，比如比较和算术运算。操作符用于在SQL语句里指定条件，还可以联接一个语句里的多个条件。

本章要介绍的操作符包括：

比较操作符；

逻辑操作符；  
求反操作符；  
算术操作符。

## [8.2 比较操作符](#)

比较操作符用于在SQL语句里对单个值进行测试。这里要介绍的比较操作符包括=、<>、<和>。

这些操作符用于测试：

相等；  
不相等；  
小于；  
大于。

下面的小节会介绍这些比较操作符的含义与用法。

### [8.2.1 相等](#)

相等操作符在SQL语句里比较一个值与另一个值，等号（=）表示相等。在进行相等比较时，被比较的值必须完全匹配，否则就不会返回数据。如果相等比较过程中的两个值相等，那么这个比较的返回值就是TRUE，否则就是FALSE。这个布尔值（TRUE或FALSE）用于决定是否返回数据。

操作符=可以单独使用，也可以与其他操作符联合使用。请记住，字符数据的比较是否区分大小写，取决于用户RDBMS的相关设置。所以，用户务必需要了解所用数据库系统对数据的比较机制。

下面的范例表示薪水等于 20 000：

```
WHERE SALARY = '20000'
```

下面的查询会返回PROD\_ID等于 2 345的全部数据：

```
SELECT *  
FROM PRODUCTS_TBL  
WHERE PROD_ID = '2345';
```

PROD_ID	PROD_DESC	COST
2345	OAK BOOKSHELF	59.99

1 row selected.

### [8.2.2 不等于](#)

有相等，就有不相等。在SQL里表示不相等的操作符是<>（一个小于号和一个大于号）。如果两个值不相等，条件就返回TRUE，否则就返回FALSE。

注意：不相等的表示方式

另一种表示不相等的方式是!=，而且很多主要的SQL实现采用这种方式。在Microsoft SQL Server、MySQL和Oracle中，两种方式是通用的。Oracle还提供了另一种方式，即^=操作符，但并不常用，因为大部分用户还是习惯于前两种方式。

下面的范例表示薪水不等于 20 000：

```
WHERE SALARY <> '20000'
```

下面的范例显示产品标识不等于 2 345的全部产品信息：

```
SELECT *
FROM PRODUCTS_TBL
WHERE PROD_ID <> '2345';
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95
2345	OAK BOOKSHELF	59.99

11 rows selected.

再提醒一次，排序规则和系统的大小写敏感性直接决定了比较的结果。在大小写敏感的情况下，系统会认为CHAIN、Chain和chain是三个不同的值，结果也就自然会与用户的预期有所差异。

### [8.2.3 小于和大于](#)

符号<（小于）和>（大于）可以自己使用，也可以与其他操作符联合使用。

下面的范例分别表示薪水小于或大于 20 000：

```
WHERE SALARY < '20000'
WHERE SALARY > '20000'
```

在第一范例里，任何小于且不等于 20 000的值会返回TRUE，大于或等于 20 000的值会返回FALSE。

```
SELECT *
FROM PRODUCTS_TBL
WHERE COST > 20;
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
2345	OAK BOOKSHELF	59.99

2 rows selected.

在下面这个范例里，请注意值 24.99 并没有包含在结果集里，因为小于号并不包含所比较的值：

```
SELECT *
FROM PRODUCTS_TBL
WHERE COST < 29.99;
```

PROD_ID	PROD_DESC	COST
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95

9 rows selected.

#### [8.2.4 比较操作符的组合](#)

等号可以与小于号和大于号联合使用。

下面的范例表示薪水小于或等于 20 000：

```
WHERE SALARY <= '20000'
```

下面的范例表示薪水大于或等于 20 000:

```
WHERE SALARY >= '20000'
```

小于等于 20 000的值包括 20 000本身及任何小于 20 000的值，在这个范围内的值会返回TRUE，大于 20 000的值会返回FALSE。大于等于操作也同样包含 20 000这个值本身。

```
SELECT *  
FROM PRODUCTS_TBL  
WHERE COST <= 29.99;
```

PROD_ID	PROD_DESC	COST
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95
11235	WITCH COSTUME	29.99

10 rows selected.

### [8.3 逻辑操作符](#)

逻辑操作符用于对SQL关键字而不是符号进行比较。下面要介绍的逻辑操作符包括:

IS NULL;

BETWEEN;

IN;



LIKE;  
EXISTS;  
UNIQUE;  
ALL和ANY。

### 8.3.1 IS NULL

这个操作符用于与NULL值进行比较。举例来说，对表EMPLOYEE\_TBL里的PAGER字段搜索NULL值，就可以找到没有寻呼机的雇员。

下面是与NULL值进行比较的一个范例，这次是对薪水进行比较：

```
WHERE SALARY IS NULL
```

下面的范例展示如何从雇员表里找到没有寻呼机的全部雇员：

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME, PAGER
FROM EMPLOYEE_TBL
WHERE PAGER IS NULL;
```

```
EMP_ID    LAST_NAM FIRST_NA PAGER
-----
311549902 STEPHENS TINA
442346889 PLEW      LINDA
220984332 WALLACE  MARIAH
443679012 SPURGEON TIFFANY
```

```
4 rows selected.
```

请注意，单词null与NULL值是不同的。观察下面这个范例：

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME, PAGER
FROM EMPLOYEE_TBL
WHERE PAGER = 'NULL';
```

```
no rows selected.
```

### 8.3.2 BETWEEN

操作符BETWEEN用于寻找位于一个给定最大值和最小值之间的值，这个最大值和最小值是包含在内的。

下面的范例表示薪水在 20 000与 30 000之间，而且包含 20 000和30 000:

```
WHERE SALARY BETWEEN '20000' AND '30000'
```

注意：适当地使用**BETWEEN**

BETWEEN是包含边界值的，所以查询结果里会包含指定的最大值和最小值。

下面的范例表示价格在\$5.95与\$14.50之间的产品:

```
SELECT *  
FROM PRODUCTS_TBL  
WHERE COST BETWEEN 5.95 AND 14.5;
```

PROD_ID	PROD_DESC	COST
222	PLASTIC PUMPKIN 18 INCH	7.75
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
1234	KEY CHAIN	5.95

```
4 rows selected.
```

可以看出，值5.95和14.5也是包含在内的。

### 8.3.3 IN

操作符 IN 用于把一个值与一个指定列表进行比较，当被比较的值至少与列表中的一个值相匹配时，它会返回TRUE。

下面的范例表示薪水必须等于 20 000、30 000或40 000中的一个

值：

```
WHERE SALARY IN('20000', '30000', '40000')
```

下面的范例展示利用操作符IN获取标识在指定范围内的产品记录：

```
SELECT *  
FROM PRODUCTS_TBL  
WHERE PROD_ID IN ('13', '9', '87', '119');
```

PROD_ID	PROD_DESC	COST
119	ASSORTED MASKS	4.95
87	PLASTIC SPIDERS	1.05
9	CANDY CORN	1.35
13	FALSE PARAFFIN TEETH	1.1

4 rows selected.

使用操作符IN可以得到与操作符OR一样的结果，但它的速度更快。

### [8.3.4 LIKE](#)

操作符LIKE利用通配符把一个值与类似的值进行比较，通配符有两个：

百分号（%）；

下划线（\_）。

百分号代表零个、一个或多个字符，下划线代表一个数字或字符。这些符号可以复合使用。

下面的条件匹配任何以200开头的值：

```
WHERE SALARY LIKE '200%
```

下面的条件匹配任何包含200（在任意位置）的值：

```
WHERE SALARY LIKE '%200%'
```

下面的条件匹配第二和第三个字符是0的值：

```
WHERE SALARY LIKE '_00%'
```

下面的条件匹配以2开头，而且长度至少为3的值：

```
WHERE SALARY LIKE '2_%_ %'
```

下面的条件匹配以2结尾的值：

```
WHERE SALARY LIKE '%2'
```

下面的条件匹配第二个位置为2，结尾为3的值：

```
WHERE SALARY LIKE '_2%3'
```

下面的条件匹配长度为5，以2开头，以3结尾的值：

```
WHERE SALARY LIKE '2___3'
```

下面的范例搜索产品描述以大写S结尾的记录：

```
SELECT PROD_DESC
FROM PRODUCTS_TBL
WHERE PROD_DESC LIKE '%S';
```

```
PROD_DESC
-----
LIGHTED LANTERNS
ASSORTED COSTUMES
PLASTIC SPIDERS
ASSORTED MASKS
```

```
4 rows selected.
```

下面的范例搜索产品描述中第二个字符是大写S的记录：

```
SELECT PROD_DESC
FROM PRODUCTS_TBL
WHERE PROD_DESC LIKE '_S%';
```

```
PROD_DESC
-----
ASSORTED COSTUMES
ASSORTED MASKS
```

```
2 rows selected.
```

### **8.3.5 EXISTS**

这个操作符用于搜索指定表里是否存在满足特定条件的记录。

下面的范例搜索表EMPLOYEE\_TBL里是否包含EMP\_ID为 333 333 333的记录：

```
WHERE EXISTS (SELECT EMP_ID FROM EMPLOYEE_TBL WHERE EMPLOYEE_ID
='333333333')
```

下面是一个子查询的范例（详情请见第14章）：

```

SELECT COST
FROM PRODUCTS_TBL
WHERE EXISTS ( SELECT COST
                FROM PRODUCTS_TBL
                WHERE COST > 100 );

```

No rows selected.

-----

这个操作没有选中任何一条记录，因为表里不存在价格超过100的记录。

再看下面这个例子：

```

SELECT COST
FROM PRODUCTS_TBL
WHERE EXISTS ( SELECT COST
                FROM PRODUCTS_TBL
                WHERE COST < 100 );

```

COST

-----

```

29.99
7.75
1.1
14.5
10
1.35
1.45
1.05
4.95
5.95
59.99

```

11 rows selected.

这一次显示了产品的价格，因为表里存在着价格小于100的记录。

### [8.3.6 ALL、SOME和ANY操作符](#)

操作符ALL用于把一个值与另一个集合里的全部值进行比较。

下面的范例测试薪水是否大于住在Indianapolis的全部雇员的薪水：

```
WHERE SALARY > ALL SALARY (SELECT FROM EMPLOYEE_TBL WHERE CITY =  
'INDIANAPOLIS')
```

下面的范例展示操作符ALL如何与子查询联合使用：

```
SELECT *  
FROM PRODUCTS_TBL  
WHERE COST > ALL ( SELECT COST  
                    FROM PRODUCTS_TBL  
                    WHERE COST < 10 );
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
2345	OAK BOOKSHELF	59.99

4 rows selected.

这个输出表示有4条记录的价格大于那些价格小于10的所有记录。

操作符ANY用于把一个值与另一个列表里任意值进行比较。SOME是ANY的别名，它们可以互换使用。

下面的范例测试薪水是否大于住在Indianapolis的任意一名雇员的薪水：

```
WHERE SALARY > ANY (SELECT SALARY FROM EMPLOYEE_TBL WHERE CITY =  
'INDIANAPOLIS')
```

下面的范例展示操作符ANY与子查询的联合使用：

```

SELECT *
FROM PRODUCTS_TBL
WHERE COST > ANY ( SELECT COST
                    FROM PRODUCTS_TBL
                    WHERE COST < 10 );

```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95
2345	OAK BOOKSHELF	59.99

10 rows selected.

这个输出结果中的记录比使用操作符ALL的多，因为这里只要求价格比小于10的价格中的任意一个高即可。价格为1.05的记录在此没有显示，因为它不大于比10小的价格中的任何一个。需要指出的是，ANY与IN是不同的，IN可以使用下面这样的表达式列表，而ANY不行：

```
IN (<Item#1>,<Item#2>,<Item#3>)
```

另外，在后面介绍求反操作符时，我们会看到与IN相反的是NOT IN，它相当于<>ALL，而不是<>ANY。

## 8.4 连接操作符

如果想在SQL语句里利用多个条件来缩小数据范围该怎么办呢？我们必须组合多个条件，这正是连接操作符的功能。连接操作符包括：



AND;

OR。

连接操作符让我们可以在一个SQL语句里用多个不同的操作符进行多种比较。下面将介绍它们的功能。

### 8.4.1 AND

操作符AND让我们可以在一条SQL语句的WHERE子句里使用多个条件。在使用AND时，无论SQL语句是事务操作还是查询，所有由AND连接的条件都必须为TRUE，SQL语句才会实际执行。

下面的范例表示EMPLOYEE\_ID必须匹配 333 333 333，并且薪水必须等于 20 000：

```
WHERE EMPLOYEE_ID = '333333333' AND SALARY = '20000'
```

下面的范例展示如何利用操作符AND来寻找价格在两个值之间的产品：

```
SELECT *  
FROM PRODUCTS_TBL  
WHERE COST > 10  
      AND COST < 30;
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
90	LIGHTED LANTERNS	14.5

```
2 rows selected.
```

在这个输出里显示了价格大于10且小于30的产品。

下面的语句不会返回任何数据，因为任何产品都只有一个标识：

```
SELECT *
FROM PRODUCTS_TBL
WHERE PROD_ID = '7725'
      AND PROD_ID = '2345';
```

no rows selected

### 8.4.2 OR

操作符OR可以在SQL语句的WHERE子句里连接多个条件，这时无论SQL语句是事务操作还是查询，只要OR连接的条件里有至少一个是TRUE，SQL语句就会执行。

下面的范例表示薪水必须匹配 20 000或30 000：

```
WHERE SALARY = '20000' OR SALARY = '30000'
```

下面的范例展示了操作符OR的具体应用：

```
SELECT *
FROM PRODUCTS_TBL
WHERE PROD_ID = '90'
      OR PROD_ID = '2345';
```

PROD_ID	PROD_DESC	COST
2345	OAK BOOKSHELF	59.99
90	LIGHTED LANTERNS	14.5

2 rows selected.

在这个输出结果里包含了满足任意一个条件的记录。

注意：比较操作符的灵活应用

比较操作符和逻辑操作符都可以单独或彼此复合使用。

在下面这个范例里使用了一个AND和两个OR，并且使用了圆括号来提高语句的可读性。

```

SELECT *
FROM PRODUCTS_TBL
WHERE COST > 10
      AND ( PROD_ID = '222'
            OR  PROD_ID = '90'
            OR  PROD_ID = '11235' );
PROD_ID    PROD_DESC                                COST
-----
11235      WITCH COSTUME                                29.99
90         LIGHTED LANTERNS                        14.5

2 rows selected.

```

提示：提高查询的可读性

当 SQL 语句里包含多个条件和操作符时，利用圆括号把语句按照逻辑关系进行划分可以提高语句的可读性。当然，不恰当地使用圆括号也会影响输出结果。

这个输出结果中的记录必须是价格大于10，而且产品标识必须是列出的三个标识之一。PROD\_ID为222的记录并没有返回，因为它的价格不大于10。圆括号不仅能够提高语句的可读性，还能够确保连接操作符能够正确地实现功能。在默认情况下，操作符是从左向右进行解析的。举例来说，寻找满足如下条件的记录：价格大于5，且PRODUCT\_ID是222、90、11 235或 13中的一个。先来看一看下面这个查询返回的结果：

```

SELECT *
FROM PRODUCTS_TBL
WHERE COST > 5
      AND (PROD_ID = '222'
      OR   PROD_ID = '90'
      OR   PROD_ID = '11235'
      OR   PROD_ID = '13');

```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
90	LIGHTED LANTERNS	14.50

3 rows in set

如果去掉其中的圆括号，就会发现返回的结果是不同的：

```

SELECT *
FROM PRODUCTS_TBL
WHERE COST > 5
      AND PROD_ID = '222'
      OR   PROD_ID = '90'
      OR   PROD_ID = '11235'
      OR   PROD_ID = '13';

```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
13	FALSE PARAFFIN TEETH	1.10
222	PLASTIC PUMPKIN 18 INCH	7.75
90	LIGHTED LANTERNS	14.50

3 rows in set

这时返回了FALSE PARAFFIN TEETH产品记录，因为现在的SQL查询条件是：PROD\_ID等于 222且COST大于 5，或者任何PROD\_ID等于 90、11 235或13的记录。在WHERE子句里正确地使用圆括号才能确保返回我们所需要的记录。如果不使用圆括号，系统通常会按照从左向

右的顺序，依次对操作符进行处理。

## 8.5 求反操作符

对于前面讨论过的所有逻辑操作符，我们都可以颠倒它们的条件要求。

操作符NOT可以颠倒逻辑操作符的含义，它可以与其他操作符构成以下几种形式：

<>, != ( NOT EQUAL);

NOT BETWEEN;

NOT IN;

NOT LIKE;

IS NOT NULL;

NOT EXISTS;

NOT UNIQUE。

下面的小节将对它们分别加以介绍，首先来看如何测试不相等。

### 8.5.1 不相等

前面已经介绍了使用操作符<>来测试不相等，在此再介绍如何测试不相等的意义在于它实际上是对相等操作符的求反。下面的范例是在某些SQL实现里测试不相等的另一种方法。

下面的范例表示薪水不等于 20 000：

```
WHERE SALARY <> '20000'  
WHERE SALARY != '20000'
```

在第二个范例里使用了惊叹号对等号操作进行求反。在某些实现里，除了可以使用标准的<>表示不相等外，还可以用惊叹号。

注意：核实惊叹号的用法

关于惊叹号的使用请查看具体实现的帮助文档。这里介绍的其他操作符在各种SQL实现里一般是相同的。

### [8.5.2 NOT BETWEEN](#)

注意：牢记**BETWEEN**的用法

操作符**BETWEEN**是包含边界值的，因此在前面这个范例里，价格等于5.95或14.50的记录就没有包含在结果里。

操作符**BETWEEN**的求反是这样的：

```
WHERE Salary NOT BETWEEN '20000' AND '30000'
```

这表示薪水不能处于 20 000与 30 000之间，而且也不包含 20 000和 30 000。再看下面这个范例：

```
SELECT *
FROM PRODUCTS_TBL
WHERE COST NOT BETWEEN 5.95 AND 14.5;
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
13	FALSE PARAFFIN TEETH	1.1
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
2345	OAK BOOKSHELF	59.99

7 rows selected.

### [8.5.3 NOT IN](#)

操作符 **IN**的求反是**NOT IN**，下面的条件表示薪水不在列表里的记录会被返回：

```
WHERE SALARY NOT IN ('20000', '30000', '40000')
```

下面的范例展示了如何使用操作符IN的求反：

```
SELECT *  
FROM PRODUCTS_TBL  
WHERE PROD_ID NOT IN (119,13,87,9);
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
6	PUMPKIN CANDY	1.45
1234	KEY CHAIN	5.95
2345	OAK BOOKSHELF	59.99

7 rows selected.

在这个输出里，标识属于操作符NOT IN之后的列表的记录没有被返回。

#### [8.5.4 NOT LIKE](#)

操作符LIKE的求反是NOT LIKE，这时只会返回不相似的值。

下面的条件表示不以200开头的值：

```
WHERE SALARY NOT LIKE '200%'
```

下面的条件表示不包含200（在任意位置）的值：

```
WHERE SALARY NOT LIKE '%200%'
```

下面的条件表示在第二个位置不包含00的值：

```
WHERE SALARY NOT LIKE '_00%'
```

下面的条件表示不是以2开始，且长度小于3的值：

```
WHERE SALARY NOT LIKE '2_ _'
```

下面的范例利用操作符NOT LIKE来显示一些值：

```
SELECT PROD_DESC  
FROM PRODUCTS_TBL  
WHERE PROD_DESC NOT LIKE 'L%';
```

```
PROD_DESC  
-----  
WITCH COSTUME  
PLASTIC PUMPKIN 18 INCH  
  
FALSE PARAFFIN TEETH  
ASSORTED COSTUMES  
CANDY CORN  
PUMPKIN CANDY  
PLASTIC SPIDERS  
ASSORTED MASKS  
KEY CHAIN  
OAK BOOKSHELF  
  
10 rows selected.
```

在这个输出结果里，不包括产品描述由字母L开始的记录。

### **8.5.5 IS NOT NULL**

操作符 IS NULL的求反是 IS NOT NULL，表示测试值不是NULL。  
下面的范例只返回NOT NULL的记录：

```
WHERE SALARY IS NOT NULL
```

下面的范例利用操作符 IS NOT NULL返回手机号不是空的雇员的记录：



```

SELECT EMP_ID, LAST_NAME, FIRST_NAME, PAGER
FROM EMPLOYEE_TBL
WHERE PAGER IS NOT NULL;

```

```

EMP_ID    LAST_NAM FIRST_NA PAGER
.....
213764555 GLASS      BRANDON 3175709980
313782439 GLASS      JACOB   8887345678

```

2 rows selected.

### [8.5.6 NOT EXISTS](#)

操作符EXISTS的求反是NOT EXISTS。

下面的范例判断EMP\_ID为 333 333 333的记录是否不在表EMPLOYEE\_TBL里：

```

WHERE NOT EXISTS (SELECT EMP_ID FROM EMPLOYEE_TBL WHERE EMP_ID =
'3333333333')

```

下面的范例展示了操作符NOT EXISTS与子查询的联合使用：

```

SELECT MAX(COST)
FROM PRODUCTS_TBL
WHERE NOT EXISTS ( SELECT COST
                    FROM PRODUCTS_TBL
                    WHERE COST > 100 );

```

```

MAX(COST)
-----
59.99

```

输出结果里显示了表里的最高价格，因为没有记录的价格高于100。

## [8.6 算术操作符](#)

算术操作符用于在SQL语句里执行算术功能，这与其他大多数语言是一样的。传统的4个算术功能是：

+（加法）；

-（减法）；

\*（乘法）；

/（除法）。

### [8.6.1 加法](#)

加法是使用加号（+）来实现的。

下面的范例把每条记录的SALARY字段和BONUS字段相加来得到合计数值：

```
SELECT SALARY + BONUS FROM EMPLOYEE_PAY_TBL;
```

下面的范例返回SALARY和BONUS字段之和大于 40 000的全部记录：

```
SELECT SALARY FROM EMPLOYEE_PAY_TBL WHERE SALARY + BONUS > '40000';
```

### [8.6.2 减法](#)

减法是使用减号（-）实现的。

下面的范例计算SALARY字段减去BONUS字段的结果：

```
SELECT SALARY - BONUS FROM EMPLOYEE_PAY_TBL;
```

下面的范例返回SALARY与BONUS字段之差大于 40 000的全部记录：

```
SELECT SALARY FROM EMPLOYEE_PAY_TBL WHERE SALARY - BONUS > '40000';
```

### 8.6.3 乘法

乘法是使用星号（\*）实现的。

下面的范例把SALARY字段乘以10：

```
SELECT SALARY * 10 FROM EMPLOYEE_PAY_TBL;
```

下面的范例返回SALARY字段乘以 10之后大于40 000的全部记录：

```
SELECT SALARY FROM EMPLOYEE_PAY_TBL WHERE SALARY * 10 > '40000';
```

下面范例里付款数额被乘以1.1，也就是把实际价格提高了10%：

```
SELECT EMP_ID, PAY_RATE, PAY_RATE * 1.1  
FROM EMPLOYEE_PAY_TBL  
WHERE PAY_RATE IS NOT NULL;
```

EMP_ID	PAY_RATE	PAY_RATE*1.1
442346889	14.75	16.225
220984332	11	12.1
443679012	15	16.5

```
3 rows selected.
```

### 8.6.4 除法

除法是使用斜线（/）实现的。

下面的范例把SALARY字段除以10：

```
SELECT SALARY / 10 FROM EMPLOYEE_PAY_TBL;
```

下面的范例返回SALARY字段大于 40 000的全部记录：

```
SELECT SALARY FROM EMPLOYEE_PAY_TBL WHERE SALARY > '40000';
```

下面的范例返回SALARY字段除以 10之后大于40 000的全部记录：

```
SELECT SALARY FROM EMPLOYEE_PAY_TBL WHERE (SALARY / 10) > '40000';
```

### 8.6.5 算术操作符的组合

算术操作符可以彼此组合使用，并且遵循基本算术运算中的优先级：首先执行乘法和除法，然后是加法和减法。用户控制算术运算次序的唯一方式是使用圆括号，圆括号里包含的表达式会被当作一个整体进行优先求值。

优先级是表达式在算术表达式里或与SQL内嵌函数结合时的求值次序。下表中的示例说明了优先级对计算结果的影响。

表达式	结果
1+1*5	6
(1+1)*5	10
10-4/2+1	9
(10-4)/(2+1)	2

从下面的范例可以看出，如果表达式中只有乘法和除法，那么有没有圆括号和它们的位置都不会影响最终结果，这时优先级没有什么影响。但是，有些SQL实现可能在这种情况下并不遵循ANSI标准，当然，这也未必。

表达式	结果
4*6/2	12
(4*6)/2	12
4*(6/2)	12

注意：确保表达式的准确性

在组合使用算术运算符时，一定要考虑到优先级的问题。语句中如果没有圆括号可能会导致不准确的结果，因为SQL语句本身的语法即使是正确的，其表示的逻辑也可能不正确。

下面是一些范例：

```
SELECT SALARY * 10 + 1000  
FROM EMPLOYEE_PAY_TBL  
WHERE SALARY > 20000;
```

```
SELECT SALARY / 52 + BONUS  
FROM EMPLOYEE_PAY_TBL;
```

```
SELECT (SALARY - 1000 + BONUS) / 52 * 1.1  
FROM EMPLOYEE_PAY_TBL;
```

下面这个范例有点复杂：

```
SELECT SALARY  
FROM EMPLOYEE_PAY_TBL  
WHERE SALARY < BONUS * 3 + 10 / 2 - 50;
```

由于没有使用圆括号，运算优先级的作用就发挥出来了，对BONUS的值进行了临时改变来进行条件判断。

## [8.7 小结](#)

本章介绍了SQL里的各种操作符，展示了它们的功能和作用，通过范例说明了这些操作符的单独使用及复合使用。介绍了基本的算术功能：加法、减法、乘法和除法。比较操作符可以测试相等、不相等、小于和大于关系，逻辑操作符包括BETWEEN、IN、LIKE、EXISTS和ALL。本章还展示了如何向SQL语句添加元素来指定更细致的条件，更好地控制SQL处理和获取数据的能力。

## [8.8 问与答](#)

问：**WHERE**子句里能包含多个**AND**吗？

答：当然可以。事实上，任何操作符都可以多次使用，举例如下：

```
SELECT SALARY
FROM EMPLOYEE_PAY_TBL
WHERE SALARY > 20000
AND BONUS BETWEEN 1000 AND 3000
AND POSITION = 'VICE PRESIDENT'
```

问：在**WHERE**子句里用单引号包围一个**NUMBER**类型的数据会怎么样呢？

答：查询仍然会执行。对于**NUMBER**类型的字段来说，单引号是没有必要的。

## **8.9 实践**

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### **8.9.1 测验**

1. 判断正误：在使用操作符**OR**时，全部条件都必须是**TRUE**。
2. 判断正误：在使用操作符**IN**时，所有指定的值都必须匹配。
3. 判断正误：操作符**AND**可以用于**SELECT**和**WHERE**子句。
4. 判断正误：操作符**ANY**可以使用一个表达式列表。
5. 操作符**IN**的逻辑求反是什么？
6. 操作符**ANY**和**ALL**的逻辑求反是什么？
7. 下面的**SELECT**语句有错吗？错在何处？
  - a.

```
SELECT SALARY
FROM EMPLOYEE_PAY_TBL
WHERE SALARY BETWEEN 20000, 30000
```

**b.**

```
SELECT SALARY + DATE_HIRE
FROM EMPLOYEE_PAY_TBL
```

**c.**

```
SELECT SALARY, BONUS
FROM EMPLOYEE_PAY_TBL
WHERE DATE_HIRE BETWEEN 2009-09-22
AND 2009-11-23
AND POSITION = 'SALES'
OR POSITION = 'MARKETING'
AND EMPLOYEE_ID LIKE '%55%
```

### 8.9.2 练习

1. 使用下面这个表CUSTOMER\_TBL，编写一条SELECT语句，选择住在Indiana、Ohio、Michigan和Illinois并且姓名以字母A或B开头的客户，返回它们的ID和姓名（以字母顺序）。

```
DESCRIBE CUSTOMER_TBL;
```

Name	Null?	Type
CUST_ID	NOT NULL	VARCHAR (10)
CUST_NAME	NOT NULL	VARCHAR (30)
CUST_ADDRESS	NOT NULL	VARCHAR (20)
CUST_CITY	NOT NULL	VARCHAR (12)
CUST_STATE	NOT NULL	VARCHAR (2)
CUST_ZIP	NOT NULL	VARCHAR (5)
CUST_PHONE		VARCHAR (10)
CUST_FAX		VARCHAR (10)

2. 使用下面这个表PRODUCTS\_TBL，编写一个SQL语句，选择产品价格在美国\$1.00与\$12.50之间的产品，返回它们的ID、描述和价格。

```
DESCRIBE PRODUCTS_TBL
```

Name	Null?	Type
PROD_ID	NOT NULL	VARCHAR (10)
PROD_DESC	NOT NULL	VARCHAR (25)
COST	NOT NULL	DECIMAL (6,2)

3. 如果在第2个练习题里使用了操作符BETWEEN，重新编写SQL语句，使用另一种操作符来得到相同的结果。如果没有使用BETWEEN，现在就来用一用。

4. 编写一个SELECT语句，返回价格小于1.00或大于12.50的产品。有两种方法可以实现。

5. 编写一个SELECT语句，从表PRODUCTS\_TBL返回以下信息：产品描述、产品价格、每个产品5%的销售税。产品列表按价格从高到低排列。

6. 编写一个SELECT语句，从表PRODUCTS\_TBL返回以下信息：产品描述、产品价格、每个产品 5%的销售税、加上销售税的总价。产品列表按价格从高到低排列。有两种方法可以实现。

7. 任选PRODUCTS\_TBL表中的3种产品。编写一个查询，返回这3种产品的相关记录。之后，再重新编写一个查询，返回除这3种产品之外的所有产品记录。在查询中，组合使用相等操作符和连接操作符。

8. 使用IN操作符重新编写练习题7中的查询。比较两种写法，哪种更高效？哪种更易读？

9. 编写一个查询，返回所有名称以P开头的产品的记录。之后，再重新编写一个查询，返回所有名称不以P开头的产品的记录。



## [第9章 汇总查询得到的数据](#)

本章的重点包括：

什么是函数

如何使用函数

何时使用函数

使用汇总函数

使用汇总函数对数据进行合计

函数得到的结果

这一章介绍SQL的汇总函数，利用它们可以实现多种功能，例如获得销售数据的最高值，或者计算某一天提交的订单总数。汇总函数的真正用途将在下一章引入GROUP BY子句后进行介绍。

### [9.1 什么是汇总函数](#)

函数是SQL里的关键字，用于对字段里的数据进行操作。函数是一个命令，通常与字段名称或表达式联合使用，处理输入的数据并产生结果。SQL 包含多种类型的函数，本章介绍汇总函数。汇总函数为SQL语句提供合计信息，比如计数、总和、平均。

本章讨论的基本汇总函数包括：

COUNT;

SUM;

MAX;

MIN;

AVG。

下面的查询显示了本章里大多数范例所使用的数据：

```
SELECT * FROM PRODUCTS_TBL;
```

PROD_ID	PROD_DESC	COST
11235	WITCH COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95
2345	OAK BOOKSHELF	59.99

11 rows selected.

下面的查询列出了表 EMPLOYEE\_TBL 里的雇员信息，注意到其中有些雇员没有呼机号。

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME, PAGER  
FROM EMPLOYEE_TBL;
```

EMP_ID	LAST_NAME	FIRST_NAME	PAGER
311549902	STEPHENS	TINA	
442346889	PLEW	LINDA	
213764555	GLASS	BRANDON	3175709980
313782439	GLASS	JACOB	8887345678
220984332	WALLACE	MARIAH	
443679012	SPURGEON	TIFFANY	

6 rows selected.

### [9.1.1 COUNT函数](#)

COUNT函数用于统计不包含NULL值的记录或字段值，在用于查询

之中时，它返回一个数值。它也可以与DISTINCT命令一起使用，从而只统计数据集里不同的记录数量。命令ALL（与DISTINCT相反）是默认的，在语句中不必明确指定。在没有指定DISTINCT的情况下，重复的行也被统计在内。使用 COUNT 函数的另一种方式是与星号配合。COUNT(\*)会统计表里的全部记录数量，包括重复的，也不管字段里是否包含NULL值。

注意：**DISTINCT**命令只能在特定情况下使用

DISTINCT命令不能与COUNT(\*)一起使用，只能用于COUNT（column\_name）。

COUNT函数的语法如下所示：

```
COUNT [ ( * ) | ( DISTINCT | ALL ) ] ( COLUMN NAME )
```

下面的范例统计全部雇员ID：

```
SELECT COUNT(EMPLOYEE_ID) FROM EMPLOYEE_PAY_ID
```

下面的范例只统计不相同的行：

```
SELECT COUNT(DISTINCT SALARY)FROM EMPLOYEE_PAY_TBL
```

下面的范例统计SALARY字段的全部行：

```
SELECT COUNT(ALL SALARY)FROM EMPLOYEE_PAY_TBL
```

下面的范例统计表EMPLOYEE\_TBL的全部行：

```
SELECT COUNT(*) FROM EMPLOYEE_TBL
```

下面的范例使用COUNT(\*)来获得表EMPLOYEE\_TBL里的全部记

录数量，结果是6。

```
SELECT COUNT(*)
FROM EMPLOYEE_TBL;

COUNT(*)
-----
6
```

注意：**COUNT(\*)**返回的结果稍有不同

与其他形式相比，**COUNT(\*)**返回的结果稍有不同。如果在**COUNT**函数中使用星号，将返回所有的统计数，包括重复项和**NULL**。这是一个很重要的差异。如果要统计某一字段的记录数，并且包括**NULL**，则需要使用**ISNULL**函数。

下面的范例使用**COUNT(EMP\_ID)**来统计表里雇员标识的数量，返回的结果与前一个查询一样，因为全部雇员都有一个标识号。

```
SELECT COUNT(EMP_ID)
FROM EMPLOYEE_TBL;

COUNT(EMP_ID)
-----
6
```

下面的范例使用 **COUNT(PAGER)**统计具有呼机号的雇员数量，从结果可以看出只有两名雇员有呼机号。

```
SELECT COUNT(PAGER)
FROM EMPLOYEE_TBL;

COUNT(PAGER)
-----
2
```

表**ORDERS\_TBL**的内容如下所示：

```
SELECT *
FROM ORDERS_TBL;
```

ORD_NUM	CUST_ID	PROD_ID	QTY	ORD_DATE_
56A901	232	11235	1	22-OCT-99
56A917	12	907	100	30-SEP-99
32A132	43	222	25	10-OCT-99
16C17	090	222	2	17-OCT-99
18D778	287	90	10	17-OCT-99
23E934	432	13	20	15-OCT-99
90C461	560	1234	2	

7 rows selected.

下面的范例统计表ORDERS\_TBL里不同的产品标识数量：

```
SELECT COUNT(DISTINCT PROD_ID )
FROM ORDERS_TBL;
```

```
COUNT(DISTINCT PROD_ID )
-----
6
```

PROD\_ID为222的记录在表里有两条，因此产品标识不同的记录数量只有6而不是7。

注意：数据类型不影响统计结果

COUNT函数统计的是行数，不涉及数据类型。行里可以包含任意类型的数据。

### 9.1.2 SUM函数

SUM函数返回一组记录中某一个字段值的总和。它也可以与DISTINCT一起使用，这时只会计算不同记录之和。这一般没有什么意义，因为有些记录被忽略掉了。

SUM函数的语法如下所示：

```
SUM ([ DISTINCT ] COLUMN NAME)
```

注意：**SUM**函数只能处理数值型字段

SUM 函数所处理的字段类型必须是数值型的，不能是其他数据类型的，比如字符或日期。

下面的范例计算薪水的总和：

```
SELECT SUM(SALARY) FROM EMPLOYEE_PAY_TBL
```

下面的范例计算不同薪水的总和：

```
SELECT SUM(DISTINCT SALARY) FROM EMPLOYEE_PAY_TBL
```

下面的查询从表PRODUCTS\_TBL里计算所有价格之和：

```
SELECT SUM(COST)
FROM PRODUCTS_TBL;
```

```
SUM(COST)
-----
163.07
```

下面的范例使用了 DISTINCT 命令，其结果与前例相比有所差别，这也说明了为什么SUM函数很少使用DISTINCT。

```
SELECT SUM(DISTINCT COST)
FROM PRODUCTS_TBL;
```

```
SUM(COST)
-----
72.14
```

下面的范例展示了虽然有些汇总函数要求使用数值型数据，但也有例外。这里使用了表EMPLOYEE\_TBL里的PAGER字段，说明CHAR数据是可以隐含地转换为数值类型的：

```
SELECT SUM(PAGER)
FROM EMPLOYEE_TBL;

SUM(PAGER)
-----
12063055658
```

如果数据不能隐含地转化为数值类型，其结果就是0。以LAST\_NAME字段为例：

```
SELECT SUM(LAST_NAME)
FROM EMPLOYEE_TBL;

SUM(LAST_NAME)
-----
0
```

### [9.1.3 AVG函数](#)

AVG函数可以计算一组指定记录的平均值。在与DISTINCT一起使用时，它返回不重复记录的平均值。AVG函数的语法如下所示：

```
AVG ([ DISTINCT ] COLUMN NAME)
```

注意：**AVG**函数只能处理数值型字段

AVG函数的参数必须是数值类型的。

下面的范例返回薪水的平均值：

```
SELECT AVG(SALARY) FROM EMPLOYEE_PAY_TBL
```

下面的范例返回不同薪水的平均值：

```
SELECT AVG(DISTINCT SALARY) EMPLOYEE_PAY_TBL
```

下面的范例计算表PRODUCTS\_TBL里COST字段全部值的平均值：

```
SELECT AVG(COST)
FROM PRODUCTS_TBL;
```

```
AVG(COST)
-----
13.5891667
```

注意：查询结果的取舍

在某些实现里，查询结果可能会被取舍到相应数据类型的精度。

下面的范例在一个查询里使用两个汇总函数。有些雇员是按小时拿工资的，有些是拿月薪，所以我们使用两个函数来计算PAY\_RATE和SALARY平均值。

```
SELECT AVG(PAY_RATE), AVG(SALARY)
FROM EMPLOYEE_PAY_TBL;
```

```
AVG(PAY_RATE)          AVG(SALARY)
-----
13.5833333             30000
```

#### [9.1.4 MAX函数](#)

MAX函数返回一组记录中某个字段的最大值，NULL值不在计算范围之内。DISTINCT也可以使用，但全部记录与不同记录的最大值是一样的，所以用DISTINCT没有意义。

MAX函数的语法如下所示：

```
MAX([ DISTINCT ] COLUMN NAME)
```



下面的范例返回最高薪水：

```
SELECT MAX(SALARY) FROM EMPLOYEE_PAY_TBL
```

下面的范例返回不同薪水中的最大值：

```
SELECT MAX(DISTINCT SALARY) FROM EMPLOYEE_PAY_TBL
```

下面的范例返回表PRODUCTS\_TBL里COST字段的最大值：

```
SELECT MAX(COST)
FROM PRODUCTS_TBL;

MAX(COST)
-----29.99

SELECT MAX(DISTINCT COST)
FROM PRODUCTS_TBL;

MAX(COST)
29.99
```

也可以对字符数据使用汇总函数，例如MAX和MIN。对于这种类型，排序规则再次发挥作用。通常，系统会将排序规则存入数据词典，查询结果会根据规则排序。在下面的范例中，我们对产品表的PRODUCT\_DESC列使用MAX函数：

```
SELECT MAX(PRODUCT_DESC)
FROM PRODUCTS_TBL;

MAX(PRODUCT_DESC)
-----
WITCH COSTUME
```

在这个范例中，函数根据数据词典返回了列中的最大值。

### 9.1.5 MIN函数

MIN 函数返回一组记录里某个字段的最小值，NULL 值不在计算之内。也可以使用DISTINCT，但由于全部记录与不同记录的最小值是一样的，所以用DISTINCT没有意义。

MIN函数的语法如下所示：

```
MIN([ DISTINCT ] COLUMN NAME)
```

下面的范例返回最低薪水：

```
SELECT MIN(SALARY) FROM EMPLOYEE_PAY_TBL
```

下面的范例返回不同薪水中的最小值：

```
SELECT MIN(DISTINCT SALARY) FROM EMPLOYEE_PAY_TBL
```

下面的范例返回表PRODUCTS\_TBL里COST字段的最小值：

```
SELECT MIN(COST)
FROM PRODUCTS_TBL;

MIN(COST)
-----
      1.05
SELECT MIN(DISTINCT COST)
FROM PRODUCTS_TBL;

MIN(COST)
-----
      1.05
```

警告：汇总函数与**DISTINCT**命令通常不一起使用

在汇总函数与DISTINCT命令一起使用时，查询返回的结果可能不是我们所需要的。汇总函数的目的在于根据表里的全部记录进行数据统

计。

与MAX函数类似，MIN函数也可以根据数据词典，返回字符型数据的最小值。

```
SELECT MIN(PRODUCT_DESC)
FROM PRODUCTS_TBL;

MIN(PRODUCT_DESC)
-----
ASSORTED COSTUMES
```

下面的范例使用了汇总函数和算术操作：

```
SELECT COUNT(ORD_NUM), SUM(QTY),
       SUM(QTY) / COUNT(ORD_NUM) AVG_QTY
FROM ORDERS_TBL;

COUNT(ORD_NUM)    SUM(QTY)    AVG_QTY
-----
7                  160         22.857143
```

这个语句统计了全部订单数量，统计了订购产品的总数，把这两个数值相除，就得到了每张订单上的平均产品数量。语句中还为计算创建了一个字段别名：AVG\_QTY。

## [9.2 小结](#)

汇总函数十分有用，而且用法很简单。本章介绍了如何统计字段里的值、统计表里的记录数量、获取字段的最大值和最小值、计算字段值的总和、计算字段值的平均值。记住，在使用汇总函数时，NULL值是不被计算的，除非以COUNT(\*)形式使用COUNT函数时。

汇总函数是本书中介绍的第一种SQL函数，后面会介绍更多的函数。汇总函数也可以用于分组值，详情在下一章介绍。大多数函数的语

法是类似的，而且其用法是相当容易理解的。

### [9.3 问与答](#)

问：在使用**MAX**或**MIN**函数时，为什么会忽略**NULL**值？

答：**NULL**值表示没有值。

问：在使用**COUNT**函数时，为什么数据类型是无关紧要的？

答：**COUNT**函数只统计记录数量。

### [9.4 实践](#)

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

#### [9.4.1 测验](#)

1. 判断正误：**AVG**函数返回全部行里指定字段的平均值，包括**NULL**值。

2. 判断正误：**SUM**函数用于统计字段之和。

3. 判断正误：**COUNT(\*)**函数统计表里的全部行。

4. 下面的**SELECT**语句能运行吗？如果不行，应该如何修改？

a.

```
SELECT COUNT *  
FROM EMPLOYEE_PAY_TBL;
```

b.

```
SELECT COUNT(EMPLOYEE_ID), SALARY  
FROM EMPLOYEE_PAY_TBL;
```

c.

```
SELECT MIN(BONUS), MAX(SALARY)
FROM EMPLOYEE_PAY_TBL
WHERE SALARY > 20000;
```

d.

```
SELECT COUNT(DISTINCT PROD_ID) FROM PRODUCTS_TBL;
```

e.

```
SELECT AVG(LAST_NAME) FROM EMPLOYEE_TBL;
```

f.

```
SELECT AVG(PAGER) FROM EMPLOYEE_TBL;
```

### 9.4.2 练习

1. 利用表EMPLOYEE\_TBL构造SQL语句，完成如下练习。
  - A. 平均薪水是多少？
  - B. 最高奖金是多少？
  - C. 总薪水是多少？
  - D. 最低小时工资是多少？
  - E. 表里有多少行记录？
2. 编写一个查询，来确定有多少雇员的姓以G开头？
3. 编写一个查询，来确定系统中所有订单的总额。如果每个产品的价格是\$10.00，全部订单的总额是多少？
4. 如果所有雇员的姓名按照字母表排序，那么编写一个查询，来确定第一个和最后一个雇员的姓名是什么？
5. 编写一个查询，对雇员姓名列使用AVG函数。查询语句能运行

吗？思考为什么会产生这样的结果。

## 第10章 数据排序与分组

本章的重点包括：

为何想对数据进行分组

**GROUP BY**子句

分组估值函数

分组函数的使用方法

根据字段进行分组

**GROUP BY**与**ORDER BY**

**HAVING**子句

前面介绍了如何对数据库进行查询，并且以一种有组织的方式返回数据，还介绍了如何对查询返回的数据进行排序。这一章将介绍如何把查询返回的数据划分为组来提高可读性。

### 10.1 为什么要对数据进行分组

数据分组是按照逻辑次序把具有重复值的字段进行合并。举例来说，一个数据库包含关于雇员的信息，雇员住在不同的城市里，有些雇员住在同一个城市里。我们可能需要进行一个查询，了解每个指定城市里的雇员的信息。这时就是在根据城市对雇员进行分组，并且创建一个摘要报告。

假设我们想了解每个城市的雇员的平均薪水，这时可以对**SALARY**字段使用**AVG**函数（前一章介绍的），并且使用**GROUP BY**子句把结果按照城市进行分组。

数据分组是通过在**SELECT**语句（查询）里使用**GROUP BY**子句来

实现的。上一章介绍了如何使用汇总函数，这一章将讨论如何联合使用汇总函数与GROUP BY子句，从而更高效地显示查询结果。

## 10.2 GROUP BY子句

GROUP BY子句与 SELECT语句配合使用，把相同的数据划分为组。在 SELECT语句里，GROUP BY子句在WHERE子句之后，在ORDER BY子句之前。

GROUP BY子句在查询中的位置如下所示：

```
SELECT
FROM
WHERE
GROUP BY
ORDER BY
```

下面是包含了GROUP BY子句的SELECT语句的语法：

```
SELECT COLUMN1, COLUMN2
FROM TABLE1, TABLE2
WHERE CONDITIONS
GROUP BY COLUMN1, COLUMN2
ORDER BY COLUMN1, COLUMN2
```

刚接触GROUP BY子句的时候，要养成按顺序书写的习惯，以确保逻辑正确。GROUP BY子句对CPU的运行效率有很大影响，如果我们不对提供给它的数据进行约束，那么后期很可能需要删除大量的无用数据。所以，需要使用WHERE子句来缩小数据范围，从而确保对有用的数据进行分组。

这里也可以加入ORDER BY子句，但RDBMS通常会使用GROUP BY子句中的列序对返回结果进行排序，本章后续内容将对此进行深入介绍。所以，除非用户对返回值的顺序有特殊要求，否则一般不会使用

ORDER BY子句。但也有一些情况需要ORDER BY子句，比如用户在SELECT语句中、GROUP BY子句外使用了汇总函数，或者用户的RDBMS与相关标准有微小差异等。

下面的小节介绍GROUP BY子句在各种场合使用的范例。

### [10.2.1 分组函数](#)

典型的分组函数——也就是用于GROUP BY子句对数据进行划分的函数——包括AVG、MAX、MIN、SUM和COUNT。它们是第9章介绍的汇总函数，当时它们是用於单个值，现在它们将用于分组值。

### [10.2.2 对选中的数据进行分组](#)

数据分组是个简单的过程。被选中的字段（查询中SELECT之后的字段列表）才能在GROUP BY子句里引用；如果字段在SELECT语句里找不到，就不能用于GROUP BY子句。这当然是合乎逻辑的——如果数据根本就不显示，我们如何对其进行分组呢？

达到要求的字段名称必须出现在GROUP BY子句里。在GROUP BY子句里可以使用字段名称，也可以使用一个整数来代表字段，具体情况稍后介绍。在对数据进行分组时，分组字段的次序不一定要与SELECT子句里的字段次序相同。

### [10.2.3 创建分组和使用汇总函数](#)

SELECT语句在使用GROUP BY子句时必须满足一定条件。特别是被选中的字段必须出现在GROUP BY子句里，除了汇总函数。GROUP BY子句里的字段不必与SELECT子句里的字段具有相同的次序。只要SELECT子句的字段名称是符合条件的，它的名称就必须出现在GROUP BY子句里，下面来介绍一些使用GROUP BY子句的语法范例。

下面的SQL语句从表EMPLOYEE\_TBL里选择字段EMP\_ID和CITY，并且对返回的数据先根据CITY，再根据EMP\_ID进行分组：



```
SELECT EMP_ID, CITY
FROM EMPLOYEE_TBL
GROUP BY CITY, EMP_ID;
```

下面的SQL语句返回EMP\_ID和SALARY字段的总和，然后根据薪水和雇员ID对数据进行分组：

```
SELECT EMP_ID, SUM(SALARY)
FROM EMPLOYEE_PAY_TBL
GROUP BY SALARY, EMP_ID;
```

下面的SQL语句从表EMPLOYEE\_TBL里返回全部薪水的总和：

```
EMPLOYEE_PAY_TBL:

SELECT SUM(SALARY) AS TOTAL_SALARY
FROM EMPLOYEE_PAY_TBL;

TOTAL_SALARY
90000.00

1 row selected
```

下面的SQL语句返回不同薪水的总和：

```
SELECT SUM(SALARY)
FROM EMPLOYEE_PAY_TBL
GROUP BY SALARY;

SUM(SALARY)
(null)
20000.00
30000.00
40000.00

4 rows selected
```

下面是使用一些实际数据的范例。在第一个范例里，我们可以看到表EMPLOYEE\_TBL里包含3个不同的城市：

```
SELECT CITY
FROM EMPLOYEE_TBL;
CITY
-----
GREENWOOD
INDIANAPOLIS
WHITELAND
INDIANAPOLIS
INDIANAPOLIS
INDIANAPOLIS

6 rows selected.
```

注意：**GROUP BY**子句中字段次序的特殊意义

注意观察SELECT语句里字段的次序，与GROUP BY子句里字段的次序进行对比。

下一个范例统计每个城市的记录数量。这时会分别看到每个不同城市的记录总和，因为其中使用了GROUP BY子句：

```
SELECT CITY, COUNT(*)
FROM EMPLOYEE_TBL
GROUP BY CITY;

CITY                COUNT(*)
-----
GREENWOOD           1
INDIANAPOLIS        4
WHITELAND            1

3 rows selected.
```

下面的查询针对一个临时表，它是基于表 EMPLOYEE\_TBL 和 EMPLOYEE\_PAY\_TBL创建的。稍后我们就会介绍如何在一个查询中联合使用两个表。

```
SELECT *
FROM EMP_PAY_TMP;
```

CITY	LAST_NAM	FIRST_NA	PAY_RATE	SALARY
-----	-----	-----	-----	-----
GREENWOOD	STEPHENS	TINA		30000
INDIANAPOLIS	PLEW	LINDA	14.75	
WHITELAND	GLASS	BRANDON		40000
INDIANAPOLIS	GLASS	JACOB		20000
INDIANAPOLIS	WALLACE	MARIAH	11	
INDIANAPOLIS	SPURGEON	TIFFANY	15	

6 rows selected.

下面的范例利用汇总函数 AVG 获得每个不同城市的平均小时工资和薪水。GREENWOOD和WHITELAND城市里没有平均小时工资，因为这两个城市里的雇员没有按小时支付的。

```
SELECT CITY, AVG(PAY_RATE), AVG(SALARY)
FROM EMP_PAY_TMP
GROUP BY CITY;
```

CITY	AVG(PAY_RATE)	AVG(SALARY)
-----	-----	-----
GREENWOOD		30000
INDIANAPOLIS	13.5833333	20000
WHITELAND		40000

3 rows selected.

下面的范例组合多种查询元素来返回分组的数据。我们只想返回 INDIANPOLIS 和WHITELAND的平均小时工资和薪水。这时只能基于 CITY进行分组，因为要对其他列使用汇总函数。最后对结果进行排序，首先是2，然后是3，即先是平均小时工资，然后是平均薪水。仔细研究下面的语句和输出结果。

```

SELECT CITY, AVG(PAY_RATE), AVG(SALARY)
FROM EMP_PAY_TMP
WHERE CITY IN ('INDIANAPOLIS','WHITELAND')
GROUP BY CITY
ORDER BY 2,3;

```

CITY	AVG(PAY_RATE)	AVG(SALARY)
INDIANAPOLIS	13.5833333	20000
WHITELAND		40000

具体数值在排序时位于NULL值之前，因此首先输出的是 INDIANAPOLIS的记录。这里没有选择 GREENWOOD 字段，否则它的记录会显示在 WHITELAND 之前，因为GREENWOOD的平均薪水是 \$30 000（ORDER BY子句里第二排序是平均薪水）。

本小节最后一个范例组合使用MAX、MIN函数与GROUP BY子句：

```

SELECT CITY, MAX(PAY_RATE), MIN(SALARY)
FROM EMP_PAY_TMP
GROUP BY CITY;

```

CITY	MAX(PAY_RATE)	MIN(SALARY)
GREENWOOD		30000
INDIANAPOLIS	15	20000
WHITELAND		40000

3 rows selected.

#### [10.2.4 以整数代表字段名称](#)

像ORDER BY子句一样，GROUP BY子句里也可以用整数代表字段名称。下面就是这样一个范例：

```
SELECT YEAR (DATE_HIRE) as YEAR_HIRED, SUM (SALARY)
FROM EMPLOYEE_PAY_TBL
GROUP BY 1;
```

YEAR_HIRED	SUM (SALARY)
-----	-----
1999	40000.00
2000	
2001	
2004	30000.00
2006	
2007	20000.00

```
6 rows selected.
```

这个 SQL 语句返回雇员薪水的总和，根据雇员参加工作的年份进行分组。GROUP BY 子句作用于整个结果集，其分组次序是 1，代表 YEAR (DATE\_HIRE)。

### **10.3 GROUP BY 与 ORDER BY**

GROUP BY 和 ORDER BY 的相同之处在于它们都是对数据进行排序。ORDER BY 子句专门用于对查询得到的数据进行排序，GROUP BY 子句也把查询得到的数据排序为适当分组的数据，因此，GROUP BY 子句也可以像 ORDER BY 子句那样用于数据排序。

用 GROUP BY 子句实现排序操作的区别与缺点是：

所有被选中的、非汇总函数的字段必须列在 GROUP BY 子句里；

除非需要使用汇总函数，否则使用 GROUP BY 子句进行排序通常是没有必要的。

下面的范例使用 GROUP BY 子句代替 ORDER BY 子句实现排序操作：

```

SELECT LAST_NAME, FIRST_NAME, CITY
FROM EMPLOYEE_TBL
GROUP BY LAST_NAME;

SELECT LAST_NAME, CITY
      *
ERROR at line 1:
ORA-00979: not a GROUP BY expression

```

注意：错误信息的返回方式不同

不同的SQL实现返回错误信息的方式会有所不同。

在这个范例里，Oracle数据库返回一条错误信息，表示LAST\_NAME不是一个GROUP BY 表达式。记住，SELECT 语句里列出的全部字段，除了汇总字段（使用汇总函数的）之外，全部都要出现在GROUP BY子句里。

下面的范例在GROUP BY子句里添加了完整的字段列表，从而解决了出现的问题：

```

SELECT LAST_NAME, FIRST_NAME, CITY
FROM EMPLOYEE_TBL
GROUP BY LAST_NAME, FIRST_NAME, CITY;

```

LAST_NAME	FIRST_NAME	CITY
-----	-----	-----
GLASS	BRANDON	WHITELAND
GLASS	JACOB	INDIANAPOLIS
PLEW	LINDA	INDIANAPOLIS
SPURGEON	TIFFANY	INDIANAPOLIS
STEPHENS	TINA	GREENWOOD
WALLACE	MARIAH	INDIANAPOLIS

6 rows selected.

这个范例从同一个表里选择相同的字段，但在GROUP BY子句里列出了SELECT子句里包含的全部字段，这时输出结果会依次按LAST\_NAME、FIRST\_NAME和CITY进行排序。虽然使用 ORDER BY

子句能够更轻松地得到这种输出结果，但本例可以帮助我们更好地理解 GROUP BY 子句的工作方式，体会它必须首先对数据进行排序才能实现分组。

下面的范例对于表 EMPLOYEE\_TBL 执行 SQL 语句，使用 GROUP BY 语句根据 CITY 进行排序：

```
SELECT CITY, LAST_NAME
FROM EMPLOYEE_TBL
GROUP BY CITY, LAST_NAME;
```

CITY	LAST_NAME
-----	-----
GREENWOOD	STEPHENS
INDIANAPOLIS	GLASS
INDIANAPOLIS	PLEW
INDIANAPOLIS	SPURGEON
INDIANAPOLIS	WALLACE
WHITELAND	GLASS

6 rows selected.

注意这个范例里数据的次序，以及每个城市里 LAST\_NAME 的次序。下面范例将统计表 EMPLOYEE\_TBL 里的全部记录，结果会按照 CITY 进行分组，但是按每个城市的雇员数量进行排序。

```
SELECT CITY, COUNT(*)
FROM EMPLOYEE_TBL
GROUP BY CITY
ORDER BY 2,1;
```

CITY	COUNT(*)
-----	-----
GREENWOOD	1
WHITELAND	1
INDIANAPOLIS	4

3 rows selected.

注意观察结果显示的次序，它首先按照每个城市的雇员数量进行排序，然后才是按城市进行排序。前两个城市的统计数量都是 1，这对应于ORDER BY子句里的第一个表达式，所以这时再根据城市进行排序，GREENWOOD位于WHITELAND之前。

虽然GROUP BY和ORDER BY具有类似的功能，但它们有一个重要区别。GROUP BY子句用于对相同的数据进行分组，而ORDER BY子句基本上只用于让数据形成次序。GROUP BY和ORDER BY可以用于同一个SELECT语句里，但必须遵守一定的次序。

提示：不能在视图中使用**ORDER BY**子句

GROUP BY 子句可以用于在 CREATE VIEW 语句里进行数据排序，而ORDER BY子句不行。CREATE VIEW语句将在第 20章介绍。

## **10.4 CUBE和ROLLUP语句**

在某些情况下，对分组数据进行小计是很有用的。例如，用户既要分析各种产品每年分别在不同国家的销售数据，也需要看到每年在每个国家所有产品的销售数据总额。ANSI SQL提供了CUBE和ROLLUP语句来解决这类问题。

ROLLUP语句可以用来进行小计，即在全部分组数据的基础上，对其中的一部分进行汇总。其ANSI语法结构如下：

```
GROUP BY ROLLUP(ordered column list of grouping sets)
```

ROLLUP 语句的工作方式是这样的，在完成了基本的分组数据汇总以后，按照从右向左的顺序，每次去掉字段列表中的最后一个字段，再对剩余的字段进行分组统计，并将获得的小计结果插入返回表中，被去掉的字段位置使用NULL填充。最后，再对全表进行一次统计，所有字段位置均使用NULL填充。Microsoft SQL Server和Oracle使用ANSI标准



语法，但MySQL的语法结构稍有不同：

```
GROUP BY order column list of grouping sets WITH ROLLUP
```

下面首先来看一个简单的GROUP BY语句返回的结果，其中我们根据城市和邮编来获得平均工资：

```
SELECT CITY, ZIP, AVG(PAY_RATE), AVG(SALARY)
FROM EMPLOYEE_TBL E
INNER JOIN EMPLOYEE_PAY_TBL P
ON E.EMP_ID=P.EMP_ID
GROUP BY CITY, ZIP
ORDER BY CITY, ZIP;
```

CITY	ZIP	AVG(PAY_RATE)	AVG(SALARY)
GREENWOOD	47890	NULL	40000
INDIANAPOLIS	45734	NULL	20000
INDIANAPOLIS	46224	14.75	NULL
INDIANAPOLIS	46234	15.00	NULL
INDIANAPOLIS	46741	11.00	NULL
WHITELAND	47885	NULL	30000

6 rows selected.

下面的范例使用了ROLLUP语句来获得小计数据：

```
SELECT CITY, ZIP, AVG(PAY_RATE), AVG(SALARY)
FROM EMPLOYEE_TBL E
INNER JOIN EMPLOYEE_PAY_TBL P
ON E.EMP_ID=P.EMP_ID
GROUP BY ROLLUP(CITY, ZIP);
```

CITY	ZIP	AVG(PAY_RATE)	AVG(SALARY)
GREENWOOD	47890	NULL	40000
GREENWOOD	NULL	NULL	40000
INDIANAPOLIS	45734	NULL	20000
INDIANAPOLIS	46224	14.75	NULL
INDIANAPOLIS	46234	15.00	NULL
INDIANAPOLIS	46741	11.00	NULL
INDIANAPOLIS	NULL	13.58	20000
WHITELAND	47885	NULL	30000
WHITELAND	NULL	NULL	30000
NULL	NULL	13.58	30000

10 rows selected.

注意观察返回结果，我们在完成了基本的分组数据汇总以后，去掉了最后一个字段（邮编），并根据剩余的字段（城市），再次进行分组统计，并将结果插入返回表。最后，还对全表进行了一次统计。

CUBE语句的工作方式与此不同。它对分组列表中的所有字段进行排列组合，并根据每一种组合结果，分别进行统计汇总。最后，CUBE语句也会对全表进行统计。CUBE语句的语法结构如下：

```
GROUP BY CUBE(column list of grouping sets)
```

CUBE语句的性质独特，因此通常被用来生成交叉报表。例如，如果需要根据城市、州、地区三个字段获得销售数据的分组统计结果，GROUP BY CUBE语句会根据以下每一种字段组合进行分组汇总，并产生统计结果。

```
CITY  
CITY, STATE  
CITY, REGION  
CITY, STATE, REGION  
REGION  
STATE, REGION  
STATE  
<grand total row>
```

CUBE语句在Microsoft SQL Server和Oracle中都可以使用，但在本书成稿之时，MySQL尚不支持该语句。下面的范例演示了如何使用CUBE语句：

```

SELECT CITY, ZIP, AVG(PAY_RATE), AVG(SALARY)
FROM EMPLOYEE_TBL E
INNER JOIN EMPLOYEE_PAY_TBL P
ON E.EMP_ID=P.EMP_ID
GROUP BY CUBE(CITY, ZIP);

```

CITY	ZIP	AVG(PAY_RATE)	AVG(SALARY)
-----	-----	-----	-----
INDIANAPOLIS	45734	NULL	20000
NULL	45734	NULL	20000
INDIANAPOLIS	46224	14.75	NULL
NULL	46224	14.75	NULL
INDIANAPOLIS	46234	15.00	NULL
NULL	46234	15.00	NULL
INDIANAPOLIS	46741	11.00	NULL
NULL	46741	11.00	NULL
WHITELAND	47885	NULL	30000
NULL	47885	NULL	30000
GREENWOOD	47890	NULL	40000
NULL	47890	NULL	40000
GREENWOOD	NULL	NULL	40000
INDIANAPOLIS	NULL	13.58	20000
WHITELAND	NULL	NULL	30000
NULL	NULL	13.58	30000

16 rows selected.

从上述范例我们可以看到，由于要根据分组列表中提供的所有字段的各种组合分别进行统计汇总，使用CUBE语句要返回的记录数会大大增加。

## 10.5 HAVING子句

HAVING子句在SELECT语句里与GROUP BY子句联合使用时，用于告诉GROUP BY子句在输出里包含哪些分组。HAVING对于GROUP BY的作用相当于WHERE对于SELECT的作用。换句话说，WHERE子句设定被选择字段的条件，而HAVING子句设置GROUP BY子句形成分组的条件。因此，使用HAVING子句可以让结果里包含或是去除整组的数据。

下面是HAVING子句在查询里的位置：

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

HAVING子句必须跟在GROUP BY子句之后、在ORDER BY子句之前。

下面是SELECT语句在包含HAVING子句时的语法：

```
SELECT COLUMN1, COLUMN2
FROM TABLE1, TABLE2
WHERE CONDITIONS
GROUP BY COLUMN1, COLUMN2
HAVING CONDITIONS
ORDER BY COLUMN1, COLUMN2
```

下面的范例选择除了GREENWOOD之外所有城市的平均小时工资和薪水。输出结果按照CITY进行分组，但只显示平均薪水超过\$20 000的分组（城市），并且按照每个城市的平均薪水进行排序。

```
SELECT CITY, AVG(PAY_RATE), AVG(SALARY)
FROM EMP_PAY_TMP
WHERE CITY <> 'GREENWOOD'
GROUP BY CITY
HAVING AVG(SALARY) > 20000
ORDER BY 3;
```

CITY	AVG(PAY_RATE)	AVG(SALARY)
WHITELAND		40000

```
1 row selected.
```

为什么这个查询只返回了一行结果？

WHERE子句把城市GREENCITY排除在外。

INDIANAPOLIS的平均薪水只有\$20 000，没有超过\$20 000，所以也不在输出结果里。

## [10.6 小结](#)

本章介绍了如何使用GROUP BY子句对查询结果进行分组。GROUP BY子句主要与汇总函数配合使用，比如SUM、AVG、MAX、MIN和COUNT。GROUP BY的本质与ORDER BY类似，也是对查询结果进行排序。GROUP BY对结果进行逻辑上的分组排序，虽然也可以实现单纯的数据排序，但就不如使用ORDER BY方便了。

HAVING子句是GROUP BY子句的一个扩充，用于对分组添加条件。相比之下，WHERE子句用于给查询的SELECT子句添加条件。下一章将介绍一些新的函数，进一步控制查询的结果。

## [10.7 问与答](#)

问：当SELECT语句里使用ORDER BY子句时，是否一定要使用GROUP BY子句？

答：不是。GROUP BY子句完全是任选的，但它与ORDER BY配合使用时会发挥很大的作用。

问：分组值是什么？

答：以表EMPLOYEE\_TBL里的CITY字段为例。如果选择雇员的姓名与城市，然后把输出按照城市进行分组，那么相同的城市就会被分在一组。

问：如果想利用GROUP BY子句根据某字段进行分组，该字段是否一定要出现在SELECT语句里？

答：是，字段必须出现在SELECT语句里，GROUP BY子句才能使

用它。

## **10.8 实践**

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### **10.8.1 测验**

1. 下面的SQL语句能正常执行吗？

**a.**

```
SELECT SUM(SALARY), EMP_ID  
FROM EMPLOYEE_PAY_TBL  
GROUP BY 1 and 2;
```

**b.**

```
SELECT EMP_ID, MAX(SALARY)  
FROM EMPLOYEE_PAY_TBL  
GROUP BY SALARY, EMP_ID;
```

**c.**

```
SELECT EMP_ID, COUNT(SALARY)  
FROM EMPLOYEE_PAY_TBL  
ORDER BY EMP_ID  
GROUP BY SALARY;
```

**d.**

```
SELECT YEAR (DATE_HIRE) AS YEAR_HIRED, SUM (SALARY)
FROM EMPLOYEE_PAY_TBL
GROUP BY 1
HAVING SUM (SALARY) > 20000;
```

2. 判断正误：在使用HAVING子句时一定也要使用GROUP BY子句。
3. 判断正误：下面的SQL语句返回分组的薪水总和：

```
SELECT SUM (SALARY)
FROM EMPLOYEE_PAY_TBL;
```

4. 判断正误：被选中的字段在GROUP BY子句里必须以相同次序出现。
5. 判断正误：HAVING子句告诉GROUP BY子句要包括哪些分组。

### 10.8.2 练习

1. 运行数据库，输入如下查询来显示表EMPLOYEE\_TBL里的全部城市：

```
SELECT CITY
FROM EMPLOYEE_TBL;
```

2. 输入如下查询，把结果与练习1的结果进行比较：

```
SELECT CITY, COUNT (*)
FROM EMPLOYEE_TBL
GROUP BY CITY;
```

3. HAVING子句与WHERE子句的相似之处在于都可以指定返回数据的条件。WHERE子句是查询的主过滤器，而HAVING子句是在

GROUP BY子句对数据进行分组之后进行过滤。输入如下查询来了解HAVING子句的工作方式：

```
SELECT CITY, COUNT(*)  
FROM EMPLOYEE_TBL  
GROUP BY CITY  
HAVING COUNT(*) > 1;
```

4. 修改练习3里的查询，把结果按降序排序，也就是数值从大到小。
5. 编写一个查询，从表EMPLOYEE\_PAY\_RATE里列出每个城市的平均税率和工资。
6. 编写一个查询，从表EMPLOYEE\_PAY\_RATE里列出城市平均薪水高于\$20 000的每个城市的平均薪水。

## [第11章 调整数据的外观](#)

本章的重点包括：

字符函数简介

如何及何时使用字符函数

ANSI SQL函数范例

常见实现的特定函数范例

转换函数概述

如何及何时使用转换函数

本章介绍如何使用函数来调整输出结果的外观，有些是 ANSI 标准函数，有些是基于该标准的函数，还有一些是由主要的SQL实现所使用的函数。

注意：**ANSI**标准并不是绝对不变的

书中介绍的ANSI概念只是概念而已。ANSI规定的标准只是对如何



在关系型数据库里使用 SQL 的一个方针，因此书中介绍的某些函数与用户所用SQL 实现里的不一定相同。它们的概念是相同的，工作方式一般也是一样的，但函数名称和实际的语法可能不同。

### 11.1 ANSI字符函数

字符函数用于在 SQL 里以不同于存储方式的格式来表示字符串。本章的第一部分讨论ANSI的字符函数概念，第二部分介绍使用不同SQL实现的函数用于实际操作。最常用的ANSI字符函数主要用于进行串接、子串和TRANSLATE等操作。

串接就是把两个单独的字符串组合为一个。举例来说，可以把个人的姓和名串接在一起形成一个字符串来表示完整的姓名。

JOHN与SMITH串接起来就得到 JOHN SMITH。

子串的概念就是从字符串里提取一部分。比如下面的值都是 JOHNSON的子串：

J;

JOHN;

JO;

ON;

SON。

TRANSLATE函数用于逐字符地把一个字符串变换为另一个，它通常有3个参数：要被转换的字符串、要转换的字符列表、代入字符的列表。稍后将介绍一些实际的范例。

### 11.2 常用字符函数

字符函数主要用于对字段里的字符串或值进行比较、连接、搜索、提取片断等，可用的字符函数有很多。

下面的小节介绍当前主要SQL厂商对ANSI概念的实现，包括Microsoft SQL Server、MySQL和Oracle。

### 11.2.1 串接函数

串接及其他一些函数在不同实现里略有不同。下面的范例展示了在Oracle和SQL Server里的串接操作。

假设要把JOHN和SON串接起来形成JOHNSON。在Oracle里的代码是这样的：

```
SELECT 'JOHN' || 'SON'
```

在SQL Server里的代码是这样的：

```
SELECT 'JOHN' + 'SON'
```

在MySQL里的代码是这样的：

```
SELECT CONCAT('JOHN' , 'SON')
```

总的来说，串接操作在Oracle里的语法是：

```
COLUMN_NAME || [ ' ' || ] COLUMN_NAME [ COLUMN_NAME ]
```

在SQL Server里的语法是：

```
COLUMN_NAME + [ ' ' + ] COLUMN_NAME [ COLUMN_NAME ]
```

在MySQL里的语法是：

```
CONCAT(COLUMN_NAME , [ ' ' , ] COLUMN_NAME [ COLUMN_NAME ])
```

MySQL 和 Oracle 中都有串接函数，用来把两个字符串连接起来，其作用相当于 SQL Server中的“+”和Oracle中的“||”。区别在于，Oracle中

的串接函数只能用于两个字符串，而 MySQL 中的串接函数可以连接多个字符串。需要注意的一点是，串接函数用于连接字符串，如果要连接数字，则需要将数字首先转换为字符串。遗憾的是，Microsoft SQL Server不支持串接函数。以下是进行串接操作的一些范例。

下面的 SQL Server语句把城市与州字段的值串接在一起，并且在两个值之间放置一个逗号：

```
SELECT CITY + STATE FROM EMPLOYEE_TBL;
```

下面的Oracle语句把城市与州字段的值串接在一起，并且在两个值之间放置一个逗号：

```
SELECT CITY || ', ' || STATE FROM EMPLOYEE_TBL;
```

这个操作在Oracle中无法使用串接函数完成，因为它连接了多个字符串。

注意：对字符串使用引号

注意前面这个 SQL 语句里单引号与逗号的使用。绝大多数字符和符号都可以被包围在单引号里。有些实现可能使用双引号来表示直义字符串。

下面的 SQL Server语句把城市与州字段的值串接在一起，并且在两个值之间放置一个空格：

```
SELECT CITY + ' ' + STATE FROM EMPLOYEE_TBL;
```

下面的SQL Server语句把个人的姓和名串接在一起，并且在两个值之间放置一个逗号：

```
SELECT LAST_NAME + ', ' + FIRST_NAME NAME
FROM EMPLOYEE_TBL;
```

NAME

-----

STEPHENS, TINA

PLEW, LINDA

GLASS, BRANDON

GLASS, JACOB

WALLACE, MARIAH

SPURGEON, TIFFANY

6 rows selected.

### 11.2.2 TRANSLATE函数

TRANSLATE函数搜索字符串里的字符并查找特定的字符，标记找到的位置，然后用替代字符串里对应的字符替换它。其语法如下所示：

```
TRANSLATE(CHARACTER SET, VALUE1, VALUE2)
```

下面的SQL语句把字符串里每个I都替换为A，每个N都替换为B，每个D都替换为C：

```
SELECT TRANSLATE (CITY, 'IND', 'ABC' FROM EMPLOYEE_TBL) CITY_TRANSLATION
```

下面的范例把TRANSLATE用于实际的数据：

```
SELECT CITY, TRANSLATE(CITY, 'IND', 'ABC')
FROM EMPLOYEE_TBL;
```

CITY	CITY_TRANSLATION
GREENWOOD	GREEBWOOC
INDIANAPOLIS	ABCAABAPOLAS
WHITELAND	WHATELABC
INDIANAPOLIS	ABCAABAPOLAS
INDIANAPOLIS	ABCAABAPOLAS
INDIANAPOLIS	ABCAABAPOLAS

6 rows selected.

在这个范例里，所有的I都被替换为A、N替换为B、D替换为C。在INDIANAPOLIS里，IND被替换为ABC，在GREENWOOD里，D被替换为C。WHITELAND的替换也是如此。

MySQL和Oracle都支持使用TRANSLATE函数，但是Microsoft SQL Server还不支持。

### 11.2.3 REPLACE

REPLACE函数用于把某个字符或字符串替换为指定的一个字符（或多个字符），其使用类似于TRANSLATE函数，只是它是把一个字符或字符串替换到另一个字符串里，其语法是：

```
REPLACE('VALUE', 'VALUE', [ NULL ] 'VALUE')
```

下面的语句返回全部的名，并且把全部的T都替换为B：

```
SELECT REPLACE(FIRST_NAME, 'T', 'B') FROM EMPLOYEE_TBL
```

下面的语句返回雇员表里的全部城市，并且把城市名称里的I都替换为Z：

```
SELECT CITY, REPLACE(CITY, 'I', 'Z')
FROM EMPLOYEE_TBL;
```

CITY	REPLACE(CITY)
GREENWOOD	GREENWOOD
INDIANAPOLIS	ZNDZANAPOLZS
WHITELAND	WHZTELAND
INDIANAPOLIS	ZNDZANAPOLZS
INDIANAPOLIS	ZNDZANAPOLZS
INDIANAPOLIS	ZNDZANAPOLZS

6 rows selected.

Microsoft SQL Server、MySQL和Oracle全都支持该函数的ANSI语法结构。

#### **11.2.4 UPPER**

大多数实现都提供了控制数据大小写的函数。UPPER函数可以把字符串里的小写字母转化为大写。

语法如下所示：

```
UPPER(character string)
```

下面的SQL语句把字段里所有的字符都转化为大写：

```
SELECT UPPER(CITY)
FROM EMPLOYEE_TBL;
```

```
UPPER(CITY)
-----
GREENWOOD
INDIANAPOLIS
WHITELAND
INDIANAPOLIS
INDIANAPOLIS
INDIANAPOLIS
```

```
6 rows selected.
```

Microsoft SQL Server、MySQL和Oracle全都支持该函数。在MySQL中，还有一个UCASE函数可以实现同样的操作，由于功能相同，用户最好还是遵循ANSI标准语法。

### **11.2.5 LOWER**

与UPPER函数相反，LOWER把字符串里的大写字符转化为小写。其语法如下所示：

```
LOWER(character string)
```

下面的语句把字段里所有的字符都转化为小写：

```
SELECT LOWER(CITY)
FROM EMPLOYEE_TBL;
```

```
LOWER(CITY)
-----
greenwood
indianapolis
whiteland
indianapolis
indianapolis
indianapolis
```

```
6 rows selected.
```

Microsoft SQL Server、MySQL和Oracle全都支持该函数。与UPPER函数类似，MySQL中也存在一个LCASE函数，但用户最好还是遵循ANSI标准语法。

### **11.2.6 SUBSTR**

在大多数 SQL 实现里都有获取字符串子串的函数，但名称可能略有不同，比如 Oracle和SQL Server。

在Oracle里的语法是：

```
SUBSTR(COLUMN NAME, STARTING POSITION, LENGTH)
```

在SQL Server里的语法是：

```
SUBSTRING(COLUMN NAME, STARTING POSITION, LENGTH)
```

对于这个函数来说，这两个实现之间的唯一差别就是函数的名称。下面的SQL语句返回EMP\_ID的前3个字符：

```
SELECT SUBSTRING(EMP_ID,1,3) FROM EMPLOYEE_TBL
```

下面的SQL语句返回EMP\_ID的第4个和第5个字符：



```
SELECT SUBSTRING(EMP_ID,4,2) FROM EMPLOYEE_TBL
```

下面的SQL语句返回EMP\_ID的第6个到第9个字符:

```
SELECT SUBSTRING(EMP_ID,6,4) FROM EMPLOYEE_TBL
```

下面的范例在SQL Server和MySQL里都可以使用:

```
SELECT EMP_ID, SUBSTRING(EMP_ID,1,3)
FROM EMPLOYEE_TBL;
```

```
EMP_ID
-----
```

```
311549902 311
```

```
442346889 442
```

```
213764555 213
```

```
313782439 313
```

```
220984332 220
```

```
443679012 443
```

```
6 rows affected.
```

下面的SQL语句是用于Oracle的:

```
SELECT EMP_ID, SUBSTR(EMP_ID,1,3)
FROM EMPLOYEE_TBL;
```

```
EMP_ID
-----
```

```
311549902 311
```

```
442346889 442
```

```
213764555 213
```

```
313782439 313
```

```
220984332 220
```

```
443679012 443
```

```
6 rows selected.
```

注意：不同实现的反馈信息有所差异

注意最后两个查询的反馈信息。前一个是“6 rows affected”，后一个是“6 rows selected”。在不同的SQL实现里都会看到类似这样的差别。

### 11.2.7 INSTR

INSTR函数用于在字符串里寻找指定的字符集，返回其所在的位置。语法如下所示：

```
INSTR(COLUMN NAME, 'SET',  
[ START POSITION [ , OCCURRENCE ] ]));
```

下面的SQL语句返回表EMPLOYEE\_TBL里每个州名里字母I第一次出现的位置：

```
SELECT INSTR(STATE, 'I', 1, 1) FROM EMPLOYEE_TBL;
```

下面的SQL语句查找字母A在字段PROD\_DESC里第一次出现的位置：

```
SELECT PROD_DESC,
       INSTR(PROD_DESC, 'A', 1, 1)
FROM PRODUCTS_TBL;
```

PROD_DESC	INSTR(PROD_DESC, 'A', 1, 1)
WITCH COSTUME	0
PLASTIC PUMPKIN 18 INCH	3
FALSE PARAFFIN TEETH	2
LIGHTED LANTERNS	10
ASSORTED COSTUMES	1
CANDY CORN	2
PUMPKIN CANDY	10
PLASTIC SPIDERS	3
ASSORTED MASKS	1
KEY CHAIN	7
OAK BOOKSHELF	2

11 rows selected.

可以看到，如果字符串里不存在字母A，返回的位置值是0。

INSTR 在 MySQL 和 Oracle 中有效，但在 Microsoft SQL Server 中，则需要使用CHARINDEX函数。

### [11.2.8 LTRIM](#)

LTRIM函数是另一种截取部分字符串的方式，它与SUBSTRING属于同一家族。LTRIM用于从左剪除字符串里的字符，其语法如下所示：

```
LTRIM(CHARACTER STRING [ , 'set' ] )
```

下面的SQL语句从所有LESLIE的左侧剪除LES：

```
SELECT LTRIM(FIRST_NAME, 'LES') FROM CUSTOMER_TBL WHERE FIRST_NAME
='LESLIE';
```

下面的SQL语句返回职位以及职位字符串里从左侧剪除SALES之后的结果：

```
SELECT POSITION, LTRIM(POSITION, 'SALES')
FROM EMPLOYEE_PAY_TBL;
```

POSITION	LTRIM(POSITION,
-----	-----
MARKETING	MARKETING
TEAM LEADER	TEAM LEADER
SALES MANAGER	MANAGER
SALESMAN	MAN
SHIPPER	HIPPER
SHIPPER	HIPPER

```
6 rows selected.
```

SHIPPER里的S也被剪除掉了，虽然SHIPPER里并不包含字符串SALES。SALES里前4个字符被忽略掉了，被搜索的字符必须以相同次序出现在目标字符串里，而且必须位于目标字符串的最左侧。换句话说，LTRIM会剪除被搜索的字符串在目标字符串里最后一次出现位置之左的全部字符。

Microsoft SQL Server、MySQL和Oracle全都支持该函数。

### [11.2.9 RTRIM](#)

类似于LTRIM，RTRIM也用于剪除字符，但它是剪除字符串的右侧。其语法如下所示：

```
RTRIM(CHARACTER STRING [ , 'set' ])
```

下面的SQL语句返回名为BRANDON的，并且剪除右侧的ON，留下BRAND作为结果：

```
SELECT RTRIM(FIRST_NAME, 'ON') FROM EMPLOYEE_TBL WHERE FIRST_NAME =
'BRANDON';
```

这个SQL语句返回表PAY\_TBL里的职位列表，并且把职位字符串

最右侧的ER剪除掉：

```
SELECT POSITION, RTRIM(POSITION, 'ER')
FROM EMPLOYEE_PAY_TBL;
```

POSITION	RTRIM(POSITION,
-----	-----
MARKETING	MARKETING
TEAM LEADER	TEAM LEAD
SALES MANAGER	SALES MANAG
SALESMAN	SALESMAN
SHIPPER	SHIPP
SHIPPER	SHIPP

```
6 rows selected.
```

全部符合条件的字符串里最右侧的ER都被剪除了。

Microsoft SQL Server、MySQL和Oracle全都支持该函数。

### [11.2.10 DECODE](#)

DECODE函数不是ANSI标准里的，至少目前还不是，但它具有强大的功能。该函数主要用于Oracle和PostgreSQL。它可以在字符串里搜索一个值或字符串，如果找到了，就在结果里显示另一个字符串。

其语法如下所示：

```
DECODE(COLUMN NAME, 'SEARCH1', 'RETURN1', [ 'SEARCH2', 'RETURN2', 'DEFAULT
VALUE' ])
```

下面的查询在表 EMPLOYEE\_TBL 里搜索全部姓，如果找到 SMITH，就会在结果里显示JONES，否则就显示OTHER（语句中设置的默认值）：

```
SELECT DECODE(LAST_NAME, 'SMITH', 'JONES', 'OTHER') FROM EMPLOYEE_TBL;
```

下面的范例对表EMPLOYEE\_TBL里的CITY字段使用DECODE：

```
SELECT CITY,
       DECODE(CITY, 'INDIANAPOLIS', 'INDY',
              'GREENWOOD', 'GREEN', 'OTHER')
FROM EMPLOYEE_TBL;
```

CITY	DECODE
GREENWOOD	GREEN
INDIANAPOLIS	INDY
WHITELAND	OTHER
INDIANAPOLIS	INDY
INDIANAPOLIS	INDY
INDIANAPOLIS	INDY

6 rows selected.

从输出结果可以看到，INDIANAPOLIS显示为INDY，GREENWOOD显示为GREEN，而其他城市显示为OTHER。

## 11.3 其他字符函数

下面的小节介绍其他一些值得一提的函数，它们在主流SQL实现里也是很常见的。

### 11.3.1 LENGTH

LENGTH函数是很常见的，用于得到字符串、数字、日期或表达式的长度，单位是字节。其语法如下所示：

```
LENGTH(CHARACTER STRING)
```

下面的SQL语句返回产品描述及其长度：

```
SELECT PROD_DESC, LENGTH(PROD_DESC)
FROM PRODUCTS_TBL;
```

PROD_DESC	LENGTH(PROD_DESC)
-----	-----
WITCH COSTUME	15
PLASTIC PUMPKIN 18 INCH	23
FALSE PARAFFIN TEETH	19
LIGHTED LANTERNS	16
ASSORTED COSTUMES	17
CANDY CORN	10
PUMPKIN CANDY	13
PLASTIC SPIDERS	15
ASSORTED MASKS	14
KEY CHAIN	9
OAK BOOKSHELF	13

```
11 rows selected.
```

MySQL和Oracle都支持该函数。而Microsoft SQL Server则使用LEN函数来实现相同的功能。

### [11.3.2 IFNULL（检查NULL值）](#)

IFNULL函数用于在一个表达式是NULL时从另一个表达式获得值。它可以用于大多数数据类型，但值与替代值必须是同一数据类型。其语法如下所示：

```
IFNULL('VALUE', 'SUBSTITUTION')
```

下面的SQL语句寻找NULL值，并且用 999 999 999代替NULL值：

```
SELECT PAGER, IFNULL(PAGER,999999999)
FROM EMPLOYEE_TBL;
```

PAGER	IFNULL(PAGER,
-----	-----
	9999999999
	9999999999
3175709980	3175709980
8887345678	8887345678
	9999999999
	9999999999

6 rows selected.

只有NULL值被替换为 999 999 999。

只有MySQL支持该函数。要实现相同的功能，Microsoft SQL Server使用 ISNULL函数，而Oracle则使用COALESCE函数。

### [11.3.3 COALESCE](#)

COALESCE函数也是用指定值替代NULL值，这一点与IFNULL是一样的。其不同点在于，它可以接受一个数据集，依次检查其中每一个值，直到发现一个非NULL值。如果没有找到非NULL值，它会返回一个NULL值。

下面的范例用COALESCE函数返回BONUS、SALARY和PAY\_RATE字段里第一个非NULL值。



```
SELECT EMP_ID, COALESCE(BONUS,SALARY,PAY_RATE)
FROM EMPLOYEE_PAY_TBL;
```

EMP_ID	COALESCE(BONUS,SALARY,PAY_RATE)
213764555	2000.00
220984332	11.00
311549902	40000.00
313782439	1000.00
442346889	14.75
443679012	15.00

6 rows selected.

Microsoft SQL Server、MySQL和Oracle全都支持该函数。

### 11.3.4 LPAD

LPAD（左填充）用于在字符串左侧添加字符或空格，其语法如下所示：

```
LPAD(CHARACTER SET)
```

下面的范例在每个产品描述左侧添加句点，使其总长度达到30个字符：

```
SELECT LPAD(PROD_DESC,30,'.') PRODUCT
FROM PRODUCTS_TBL;
```

```
PRODUCT
-----
.....WITCH COSTUME
.....PLASTIC PUMPKIN 18 INCH
.....FALSE PARAFFIN TEETH
.....LIGHTED LANTERNS
.....ASSORTED COSTUMES
.....CANDY CORN
.....PUMPKIN CANDY
.....PLASTIC SPIDERS
.....ASSORTED MASKS
.....KEY CHAIN
.....OAK BOOKSHELF
```

```
11 rows selected.
```

MySQL和Oracle全都支持该函数。遗憾的是，Microsoft SQL Server中没有对应的函数。

### **11.3.5 RPAD**

RPAD（右填充）在字符串右侧添加字符或空格，其语法如下所示：

```
RPAD(CHARACTER SET)
```

下面的范例在每个产品描述的右侧添加句点，让总长度达到30个字符：

```
SELECT RPAD(PROD_DESC,30,'.') PRODUCT
FROM PRODUCTS_TBL;
```

```
PRODUCT
-----
WITCH COSTUME.....
PLASTIC PUMPKIN 18 INCH.....
FALSE PARAFFIN TEETH.....
LIGHTED LANTERNS.....
ASSORTED COSTUMES.....
CANDY CORN.....
PUMPKIN CANDY.....
PLASTIC SPIDERS.....
ASSORTED MASKS.....
KEY CHAIN.....
OAK BOOKSHELF.....
```

```
11 rows selected.
```

MySQL和Oracle全都支持该函数。遗憾的是，Microsoft SQL Server中没有对应的函数。

### 11.3.6 ASCII

ASCII函数返回字符串最左侧字符的“美国信息交换标准码（ASCII）”，其语法如下所示：

```
ASCII(CHARACTER SET)
```

下面是一些范例：

ASCII('A')返回65；

ASCII('B')返回66；

ASCII('C')返回67；

ASCII('a')返回95。

更多信息请参见[www.asciitable.com](http://www.asciitable.com)上的ASCII表。

Microsoft SQL Server、MySQL和Oracle全都支持该函数。

## [11.4 算术函数](#)

在多个不同实现之间，算术函数是相对比较标准的。算术函数可以对数据库里的值根据算术规则进行运算。

最常见的算术函数包括：

绝对值（ABS）

舍入（ROUND）

平方根（SQRT）

符号（SIGN）

幂（POWER）

上限和下限（CEIL、FLOOR）

指数（EXP）

SIN、COS、TAN

大多数算术函数的语法是：

**FUNCTION(*EXPRESSION*)**

Microsoft SQL Server、MySQL和Oracle都支持所有的算数函数。

## [11.5 转换函数](#)

转换函数把数据类型从一种转换为另一种。举例来说，我们的数据通常是以字符形式保存的，但为了计算就需要把它转换为数值。算术函数和计算不能用于以字符形式表示的数据。

下面是一些常见的数据转换：

字符到数字；

数字到字符；

字符到日期；

日期到字符。

本章将介绍前两种转换，其他的转换在第12章介绍。

### 11.5.1 字符串转换为数字

数值数据类型与字符串数据类型有两个主要的区别：

算术表达式和函数可以用于数值；

在输出结果里，数值是右对齐的，而字符串是左对齐的。

注意：转换为数值

对于要转换为数值的字符串来说，其中的字符必须是 0~9。另外加号、减号和句点可以分别用来表示正数、负数和小数。举例来说，字符串“STEVE”不能转化为数值，而个人的社会保险号码能够以字符串形式保存，并可以利用转换函数方便地转换为数值。

当字符串转化为数值时，它就具有了上述两个特点。

有些实现没有把字符串转化为数值的函数，有些有。无论是何种情况，请查看相应的文档来了解转换的语法和规则。

注意：某些实现的自动转换

有些实现在需要时会隐含进行数据类型转换，这意味着系统会自动进行转换，这时就不必使用转换函数了。详细情况请参考具体实现的帮助文档。

下面是使用Oracle转换函数的一个数值转换范例：

```
SELECT EMP_ID, TO_NUMBER(EMP_ID)
FROM EMPLOYEE_TBL;
```

EMP_ID	TO_NUMBER(EMP_ID)
-----	-----
311549902	311549902
442346889	442346889
213764555	213764555
313782439	313782439
220984332	220984332
443679012	443679012

6 rows selected.

雇员标识在转换后就变成右对齐的。

### [11.5.2 数字转换为字符串](#)

与前面的转换相比，数值转换为字符串是个完全相反的过程。

下面的范例使用SQL Server里的转换函数把数值转换为字符串：

```
SELECT PAY = PAY_RATE, NEW_PAY = STR(PAY_RATE)
FROM EMPLOYEE_PAY_TBL
WHERE PAY_RATE IS NOT NULL;
```

PAY	NEW_PAY
-----	-----
17.5	17.5
14.75	14.75
18.25	18.25
12.8	12.8
11	11
15	15

6 rows affected.

提示：不同数据类型的对齐方式不同

数据的对齐方式是判别字段数据类型的最简单方式。

下面的范例使用Oracle的函数实现完全相同的转换：

```
SELECT PAY_RATE, TO_CHAR(PAY_RATE)
FROM EMPLOYEE_PAY_TBL
WHERE PAY_RATE IS NOT NULL;
```

PAY_RATE	TO_CHAR(PAY_RATE)
17.5	17.5
14.75	14.75
18.25	18.25
12.8	12.8
11	11
15	15

6 rows selected.

## [11.6 字符函数的组合使用](#)

大多数函数可以在 SQL 语句里组合使用。如果不允许函数组合使用，SQL 就会有很大的局限性。下面的范例在一个查询里组合使用两个函数（串接和子串），把 EMP\_ID 字段分为3部分，再用短划线把它们连接起来，从而得到更清晰易读的社会保险号码。范例里使用了 CONCAT 函数来组合字符串。

```

SELECT CONCAT(LAST_NAME, ' ', FIRST_NAME) NAME,
       CONCAT(SUBSTR(EMP_ID,1,3), '- ',
              SUBSTR(EMP_ID,4,2), '- ',
              SUBSTR(EMP_ID,6,4)) AS ID
FROM EMPLOYEE_TBL;

```

NAME	ID
-----	-----
STEPHENS, TINA	311-54-9902
PLEW, LINDA	442-34-6889
GLASS, BRANDON	213-76-4555
GLASS, JACOB	313-78-2439
WALLACE, MARIAH	220-98-4332
SPURGEON, TIFFANY	443-67-9012

6 rows selected.

下面的范例使用LENGTH函数和算术运算符（+）把每个字段的姓和名的长度加在一起，然后SUM函数返回所有姓和名的长度之和。

```

SELECT SUM(LENGTH(LAST_NAME) + LENGTH(FIRST_NAME)) TOTAL
FROM EMPLOYEE_TBL;

```

TOTAL
-----
71

1 row selected.

注意：内嵌函数的处理

当SQL语句的函数内部嵌有函数时，最内层的函数首先被处理，然后从里向外依次执行各个函数。

## [11.7 小结](#)

到目前为止，我们介绍了在SQL语句（通常是个查询）里使用多种



函数来调整或强化输出结果的外观。这些函数包括字符函数、算术函数和转换函数。需要明确的是，ANSI 标准是如何实现SQL的一个方针，但没有规定准确的语法或位置限制。大多数厂商提供了标准函数，并且遵循ANSI标准，但也都有自己的函数。函数名和语法可能有所不同，但其概念都是相同的。

## [11.8 问与答](#)

问：所有的函数都符合**ANSI**标准吗？

答：不，并不是所有函数都严格遵守ANSI标准。函数像数据类型一样，经常是取决于具体实现的。大多数实现具有ANSI函数的超集，有些实现包含了广泛的函数来扩展功能，而有些则有所局限。本章展示了一些常用实现的函数范例，但在具体使用时，由于很多实现具有类似的函数（但可能略有区别），用户应该查看相应的文档来了解可用的函数及其用法。

问：在使用函数时，数据库里的数据是否实际发生了改变？

答：没有。在使用函数时，数据库里的数据没有改变。函数通常是用在查询语句里来调整输出的外观。

## [11.9 实践](#)

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### [11.9.1 测验](#)

1. 匹配函数与其描述。

## 描述 函数

- a. 从字符串里选择一部分 ||
  - b. 从字符串左侧或右侧剪切字符串 RPAD
  - c. 把全部字符都改变为大写 LPAD
  - d. 确定字符串的长度 RTRIM
  - e. 连接字符串 UPPER LTRIM LENGTH LOWER SUBSTR
2. 判断正误：在 SELECT 语句里使用函数调整数据输出外观时会影响数据库里存储的数据。
3. 判断正误：当查询里出现函数嵌套时，最外层的函数会首先被处理。

### 11.9.2 练习

1. 在mysql>提示符下输入如下命令，把每个雇员的姓和名连接起来：

```
SELECT CONCAT(LAST_NAME, ' ', FIRST_NAME)
FROM EMPLOYEE_TBL;
```

在Oracle和SQL Server中如何实现该语句？

2. 输入以下MySQL命令，显示每个雇员的完整姓名和电话区号：

```
SELECT CONCAT(LAST_NAME, ' ', FIRST_NAME), SUBSTRING(PHONE, 1, 3)
FROM EMPLOYEE_TBL;
```

在Oracle和SQL Server中如何实现该语句？

3. 编写一个SQL语句，列出雇员的电子邮件地址。电子邮件地址并不是数据库里的一个字段，雇员的电子邮件地址应该由以下形式构成：

```
FIRST.LAST@PERPTECH.COM
```

举例来说，John Smith的电子邮件地址是  
JOHN.SMITH@PERPTECH.COM。

4. 编写一个SQL语句，以如下形式列出雇员的姓名、ID和电话号码。

- a. 姓名显示为SMITH, JOHN;
- b. 雇员ID显示为999-99-9999;
- c. 电话号码显示为(999)999-9999。

## [第12章 日期和时间](#)

本章的重点包括：

理解日期和时间

日期和时间是如何存储的

典型的日期和时间格式

如何使用日期函数

如何使用日期转换

本章介绍 SQL 中的日期和时间，不仅要详细讨论 DATETIME 数据类型，还会讨论某些实现如何使用日期、如何从期望的格式中提取日期和时间，以及其他一些常见规则。

注意：**SQL**的不同实现

众所周知，SQL的实现有多种。本书介绍ANSI标准及最常见的非标准函数、命令和操作符。本书的范例使用 MySQL，但即使是在 MySQL里，日期的保存格式也有多种，用户必须查看相应的文档来了解实际情况。但无论以何种格式存储日期，SQL实现中都有转化格式的函数。

### [12.1 日期是如何存储的](#)

每个实现都有一个默认的日期和时间存储格式，但这种默认格式一般是不同的。下面的小节首先复习 DATETIME 数据格式的标准格式及其元素，然后介绍某些流行 SQL 实现中的日期和时间数据类型，包括 Oracle、MySQL和Microsoft SQL Server。

### 12.1.1 日期和时间的标准数据类型

日期和时间（DATETIME）存储的标准SQL数据类型有3种。

**DATE：**直接存储日期。DATE 的格式是 YYYY-MM-DD，范围是从 0001-01-01 到9999-12-31。

**TIME：**直接存储时间。TIME 的格式是 HH:MI:SS.nn...，范围是从 00:00:00...到23:59:61.999...。

**TIMESTAMP：**直接存储日期和时间。TIMESTAMP 的格式是 YYYY-MM-DD HH:MI:SS.nn...，范围是从 0001-01-01 00:00:00...到 9999-12-31 23:59:61.999...。

### 12.1.2 DATETIME元素

DATETIME元素是属于日期和时间的元素，包含在DATETIME定义里。下面列出了必须有的DATETIME元素及其取值范围。

DATETIME 元素	有效范围
YEAR	0001 到 9999
MONTH	01 到 12
DAY	01 到 31
HOUR	00 到 23
MINUTE	00 到 59
SECOND	00.000...到 61.999...

每一种元素都是我们日常会遇到的。秒是以小数表示的，允许表达式的值是十分之一秒、百分之一秒、毫秒等。有些人可能要问一分钟会超过60秒吗？根据ANSI标准，61.999秒用于插入或略去闰秒，而这是很

少发生的事情。不同实现中日期和时间的存储可能差别很大，用户请参考具体实现的文档。

注意：闰年由数据库处理

如果数据以DATETIME格式存储在数据库里，像闰秒和闰年这样的日期调整是由数据库在内部完成的。

12.1.3 不同实现的日期类型

像其他数据类型一样，每种实现都有自己的形式和语法来处理日期和时间。表12.1介绍了 3种实现（Microsoft SQL Server、MySQL和Oracle）的日期和时间。

表12.1 不同平台的日期类型

产品	数据类型	用途
Oracle	DATE	存储日期和时间信息
SQL Server	DATETIME	存储日期和时间信息
	SMALLDATETIME	存储日期和时间信息，但取值范围小于 DATETIME
	DATE	存储日期值
	TIME	存储时间值

续表

产品	数据类型	用途
MySQL	DATETIME	存储日期和时间信息
	TIMESTAMP	存储日期和时间信息
	DATE	存储日期值
	TIME	存储时间值
	YEAR	单字节，表示年

12.2 日期函数

日期函数在每个不同实现里是有所区别的。类似于字符串函数，日

期函数用于调整日期和时间数据的外观，以适当的方式显示日期和时间数据、进行比较、计算日期之间的间隔等。

注意：日期和时间类型在不同的实现中有所不同

每种实现都有自己的数据类型来存储日期和时间信息，但大多数实现遵循ANSI标准，也就是说日期和时间的全部元素都保存在相关的数据类型里。日期实际的存储方式是取决于具体实现的。

### 12.2.1 当前日期

有人可能已经产生问题了：如何从数据库获取当前日期呢？在很多情况下都可能需要从数据库获取当前日期，最常见的是用来与存储的日期进行比较，或是作为某种时间标记。

从根本上来说，当前日期保存在数据库所在的计算机上时，被称为系统日期。数据库通过与操作系统进行交互可以获取系统日期，从而用于自身需要或是满足数据库请求（比如查询）。

下面介绍几种不同实现里获取系统日期的一些方法。

Microsoft SQL Server使用名为GETDATE()的函数获取系统日期，其使用方法及返回值如下所示：

```
SELECT GETDATE()
```

```
Dec 31, 2010
```

MySQL使用NOW函数获取当前日期和时间。NOW被称为伪字段，因为它具有像其他字段一样的行为，能够从数据库里的任意表里被选择，但它实际上并不存在于任何表的定义里。

下面是使用MySQL语句获取当前日期和时间的范例：

```
SELECT NOW ();
```

```
31-DEC-11 13:41:45
```

Oracle使用SYSDATE函数，以下范例使用了Oracle中的DUAL表：

```
SELECT SYSDATE FROM DUAL;
```

```
31-DEC-11 13:41:45
```

### [12.2.2 时区](#)

在处理日期和时间信息时，可能要考虑时区。举例来说，美国中部时间下午6:00并不等同于澳大利亚的同一时间。另外，在使用夏时制的地区，每年都要调整两次时间。如果在维护数据时需要考虑时区问题，我们就需要处理时区和进行时间转换（如果SQL实现里有这样的函数）。

下面是一些常见时区及其缩写。

缩写	定义
AST、ADT	大西洋标准时间、大西洋白天时间
BST、BDT	白令标准时间、白令白天时间
CST、CDT	中部标准时间、中部白天时间
EST、EDT	东部标准时间、东部白天时间
GMT	格林威治标准时间
HST、HDT	阿拉斯加/夏威夷标准时间、阿拉斯加/夏威夷白天时间
MST、MDT	山区标准时间、山区白天时间
NST	纽芬兰标准时间、纽芬兰白天时间
PST、PDT	太平洋标准时间、太平洋白天时间
YST、YDT	育空标准时间、育空白天时间

下面是在某个给定时间不同时区之间的差别：

时区	时间
AST	2010 年 6 月 12 日, 1:15 pm
BST	2010 年 6 月 12 日, 6:15 am
CST	2010 年 6 月 12 日, 11:15 am
EST	2010 年 6 月 12 日, 12:15 pm
GMT	2010 年 6 月 12 日, 5:15 pm
HST	2010 年 6 月 12 日, 7:15 am
MST	2010 年 6 月 12 日, 10:15 am
NST	2010 年 6 月 12 日, 1:45 pm
PST	2010 年 6 月 12 日, 9:15 am
YST	2010 年 6 月 12 日, 8:15 am

注意：处理时区

有些实现里包含了能够处理时区的函数，但并不是所有实现都支持使用时区，实际应用时要考虑特定的实现及需求。

### [12.2.3 时间与日期相加](#)

日、月以及时间的其他组成部分可以加到日期上，从而进行日期比较或是在WHERE子句里提供更精确的条件。

DATETIME值可以增加时间间隔。根据标准的定义，时间间隔用于调整DATETIME值，如下例所示：

```
DATE '2010-12-31' + INTERVAL '1' DAY
```

```
'2011-01-01'
```

```
DATE '2010-12-31' + INTERVAL '1' MONTH
```

```
'2011-01-31'
```

下面是使用SQL Server的DATEADD函数的范例：



```
SELECT DATE_HIRE, DATEADD(MONTH, 1, DATE_HIRE)
FROM EMPLOYEE_PAY_TBL;
```

DATE_HIRE	ADD_MONTH
-----	-----
23-MAY-99	23-JUN-99
17-JUN-00	17-JUL-00
14-AUG-04	14-SEP-04
28-JUN-07	28-JUL-07
22-JUL-06	22-AUG-06
14-JAN-01	14-FEB-01

6 rows affected.

下面是使用Oracle的ADD\_MONTHS函数的范例：

```
SELECT DATE_HIRE, ADD_MONTHS(DATE_HIRE,1)
FROM EMPLOYEE_PAY_TBL;
```

DATE_HIRE	ADD_MONTH
-----	-----
23-MAY-99	23-JUN-99
17-JUN-00	17-JUL-00
14-AUG-04	14-SEP-04
28-JUN-07	28-JUL-07
22-JUL-06	22-AUG-06
14-JAN-01	14-FEB-01

6 rows selected.

在Oracle里，如果想向日期上增加一天，处理方式如下所示：

```
SELECT DATE_HIRE, DATE_HIRE + 1
FROM EMPLOYEE_PAY_TBL
WHERE EMP_ID = '311549902';
```

DATE_HIRE	DATE_HIRE
-----	-----
23-MAY-99	24-MAY-99

1 row selected.

如果想在MySQL里进行同样的查询，可以使用ANSI标准的INTERVAL命令，如下所示。如果使用像Oracle那样的方式，MySQL会把日期转换为整数再进行加法运算。

```
SELECT DATE_HIRE, DATE_ADD(DATE_HIRE, INTERVAL 1 DAY), DATE_HIRE + 1
FROM EMPLOYEE_PAY_TBL
```

```
WHERE EMP_ID = '311549902';
```

DATE_HIRE	DATE_ADD	DATE_HIRE+1
-----	-----	-----
23-MAY-99	24-MAY-99	1990524

1 row selected.

从MySQL、SQL Server和Oracle的这些范例可以看出，虽然它们从句法上都与ANSI标准有所区别，但其结果都是基于SQL标准所描述的概念。

[12.2.4 其他日期函数](#)

表 12.2列出了SQL Server、Oracle和MySQL里其他一些日期函数。

表12.2 不同平台的日期函数

产品	日期函数	用途
SQL Server	DATEPART	返回日期的某个元素的整数值
	DATENAME	返回日期的某个元素的文本值
	GETDATE()	返回系统日期
	DATEDIFF	返回两个日期之间由指定日期元素表示的间隔，比如天数、分钟数和秒数
Oracle	NEXT_DAT	返回指定日期之后的下一天（比如 FRIDAY）
	MONTHS_BETWEEN	返回两个日期之间相差的月数
MySQL	DAYNAME(date)	显示星期几
	DAYOFMONTH(date)	显示几日
	DAYOFWEEK(date)	显示星期几
	DAYOFYEAR(date)	显示一年中的第几天

## 12.3 日期转换

很多原因都会导致进行日期转换，主要用于转换定义为 DATETIME 的数据类型，或是具体实现中其他的数据类型。

进行日期转换的典型原因有：

比较不同数据类型的日期值；

把日期值格式化为字符串；

把字符串转化为日期格式。

ANSI的CAST操作符可以把一种数据类型转换为另一种，其基本语法如下所示：

```
CAST ( EXPRESSION AS NEW_DATA_TYPE )
```

下面的小节会介绍一些特定实现中的具体语法，包括：

DATETIME值里元素的表示；

日期转化为字符串；

字符串转化为日期。

### 12.3.1 日期描述

日期描述由格式元素组成，用于从数据库以期望的格式提取日期和时间信息。日期描述并不是在所有实现里都存在。

如果不使用日期描述和某种转换函数，日期和时间信息从数据库里是以默认格式提取的，如下所示：

```
2010-12-31
31-DEC-10
2010-12-31 23:59:01.11
...
```

如果我们想以如下方式显示日期该怎么办呢？

```
December 31, 2010
```

这时我们不得不把日期从DATETIME模式转化为字符串，这是由一些专用函数完成的，稍后将加以介绍。

表12.3展示了很多实现里所使用的常见日期元素，它们可以帮助我们获取适当的日期时间信息。

表12.3 常见日期元素

产品	语法	日期元素
SQL Server	yy	年
	qq	季度
	mm	月
	dy	积日（从历年的第一天累积的天数）
	wk	星期
	dw	周日
	hh	小时
	mi	分钟
	ss	秒
	ms	毫秒
Oracle	AD	公元
	AM	正午以前
	BC	公元前
	CC	世纪
	D	星期中的第几天
	DD	月份中的第几天
	DDD	年中的第几天
	DAY	拼写出来的周日（比如 MONDAY）
	Day	拼写出来的周日（比如 Monday）
	day	拼写出来的周日（比如 monday）
	DY	周日的三字母缩写（比如 MON）
	Dy	周日的三字母缩写（比如 Mon）
	dy	周日的三字母缩写（比如 mon）

续表

产品	语法	日期元素
Oracle	HH	小时
	HH12	小时
	HH24	小时（24 小时制）
	J	自公元前 4713 年 12 月 31 日起至今的日子
	MI	分钟数
	MM	月份
	MON	月份的三字母缩写（比如 JAN）
	Mon	月份的三字母缩写（比如 Jan）
	mon	月份的三字母缩写（比如 jan）
	MONTH	月份的拼写（比如 JANUARY）
	Month	月份的拼写（比如 January）
	month	月份的拼写（比如 january）
	PM	正午之后
	Q	季度数
	RM	以罗马数字表示的月份
	RR	两位数字表示的年份
	SS	秒数
	SSSSS	自午夜起累积的秒数
	SYYYYY	以符号数表示的年份，比如公元前 500 年就表示为-500
	W	月里的第几星期
	WW	年里的第几星期
	Y	年份的最后一位数字
	YY	年份的最后两位数字
	YYY	年份的最后三位数字
	YYYY	年份
	YEAR	拼写出来的年份（TWO-THOUSAND-TEN）
	Year	拼写出来的年份（Two-Thousand-Ten）
	year	拼写出来的年份（two-thousand-ten）
MySQL	SECOND	秒
	MINUTE	分钟
	HOURL	小时
	DAY	天
	MONTH	月
	YEAR	年
	MINUTE_SECOND	分和秒
	HOURL_MINUTE	小时和分
	DAY_HOURL	天和小时
	YEAR_MONTH	年和月
	HOURL_SECOND	小时、分和秒
	DAY_MINUTE	天和分钟
	DAY_SECOND	天和秒

注意：上表所列是MySQL里最通用的日期元素，不同版本的MySQL里还有其他一些可以使用的日期元素。

### 12.3.2 日期转换为字符串

日期转换为字符串是为了改变日期在查询中的输出形式，它是通过使用转换函数实现的。下面展示了两个把日期和时间数据转换为字符串的范例，首先是使用SQL Server:

```
SELECT DATE_HIRE = DATENAME(MONTH, DATE_HIRE)
FROM EMPLOYEE_PAY_TBL;
```

```
DATE_HIRE
-----
May
June
August
June
July
January
```

6 rows affected.

第二个范例是Oracle，它使用TO\_CHAR函数：

```
SELECT DATE_HIRE, TO_CHAR(DATE_HIRE, 'Month dd, yyyy') HIRE
FROM EMPLOYEE_PAY_TBL;
```

DATE_HIRE	HIRE
23-MAY-99	May 23, 1999
17-JUN-00	June 17, 2000
14-AUG-04	August 14, 2004
28-JUN-07	June 28, 2007
22-JUL-06	July 22, 2006
14-JAN-01	January 14, 2001

6 rows selected.

### 12.3.3 字符串转换为日期

下面的范例展示了一个 MySQL 实现里把字符串转化为日期格式。在转换完成之后，数据可以保存到定义为某种 DATETIME 数据类型的字段里。

```
SELECT STR_TO_DATE('01/01/2010 12:00:00 AM', '%m/%d/%Y %h:%i:%s %p') AS  
FORMAT_DATE  
FROM EMPLOYEE_PAY_TBL;
```

```
FORMAT_DATE
```

```
-----
```

```
01-JAN-10
```

```
01-JAN-10
```

```
01-JAN-10
```

```
01-JAN-10
```

```
01-JAN-10
```

```
01-JAN-10
```

```
6 rows selected.
```

有人也许会问，前例中只提供了一个日期值，为什么会显示有6条记录被选择呢？这是因为被转换的字符串来自于表 EMPLOYEE\_PAY\_TBL，而它有6行数据。

在 Microsoft SQL Server 中，我们使用 CONVERT 函数：

```
SELECT CONVERT(DATETIME, '02/25/2010 12:00:00 AM') AS FORMAT_DATE  
FROM EMPLOYEE_PAY_TBL;  
FORMAT_DATE
```

```
-----
```

```
2010-02-25 00:00:00.000
```

```
2010-02-25 00:00:00.000
```

```
2010-02-25 00:00:00.000
```

```
2010-02-25 00:00:00.000
```

```
2010-02-25 00:00:00.000
```

```
2010-02-25 00:00:00.000
```

```
6 rows selected.
```



## [12.4 小结](#)

本章介绍了DATETIME值是基于ANSI标准的，但就像很多SQL元素一样，大多数实现偏离了标准 SQL 命令的名称与语法，但其表示与操作日期和时间信息的基本概念没有改变。前一章介绍了函数在不同实现里的区别，而这一章介绍了日期和时间类型、函数和操作符之间的不同。记住，在此介绍的范例并不是在所有SQL实现里都能执行的，但其基本概念是相同的，适用于任何实现。

## [12.5 问与答](#)

问：不同实现为什么与数据类型和函数的单一标准集有所差别？

答：不同实现在数据类型和函数外观方面有所区别，主要是因为不同的厂商使用不同的方式保存数据，追求用最有效的方式提供数据检索。但是，所有实现都应该根据ANSI描述的必要元素来提供保存日期和时间的相同方式，比如年、月、日、小时、分钟、秒等。

问：如果想用不同于实现所提供的方式来保存日期和时间信息，应该怎么办呢？

答：如果把保存日期的字段定义为变长字符串类型，我们几乎可以用任何格式来保存日期。这时主要要注意的是，如果想进行日期值的比较，我们首先要把日期的字符串形式转化为DATETIME形式。

## [12.6 实践](#)

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### 12.6.1 测验

1. 系统日期和时间源自于哪里？
2. 列出DATETIME值的标准内部元素。
3. 如果公司是个国际公司，在处理日期和时间的比较与表示时，应该考虑的一个重要因素是什么？
4. 字符串表示的日期值能不能与定义为某种DATETIME类型的日期值进行比较？
5. 在SQL Server、MySQL和Oracle里，使用什么函数获取当前日期和时间？

### 12.6.2 练习

1. 在不同系统中输入以下SQL代码，从服务器显示当前日期：

```
a. MySQL : SELECT CURRENT_DATE;  
b. SQL Server : SELECT GETDATE();  
c. Oracle : SELECT SYSDATE FROM DUAL;
```

2. 输入以下SQL代码，显示每名雇员的受雇日期：

```
SELECT EMP_ID, DATE_HIRE  
FROM EMPLOYEE_PAY_TBL;
```

3. 在MySQL里，通过联合使用EXTRACT函数与MySQL日期描述，我们能够以多种格式显示日期。输入以下代码显示每名雇员的受雇年份：

```
SELECT EMP_ID, EXTRACT(YEAR FROM DATE_HIRE)  
FROM EMPLOYEE_PAY_TBL;
```

4. 在Microsoft SQL Server中运行以下代码：

```
SELECT EMP_ID, YEAR( DATE_HIRE)
FROM EMPLOYEE_PAY_TBL;
```

5. 输入以下类似MySQL实现的代码，显示当前日期和每名雇员的受雇日期：

```
SELECT EMP_ID, DATE_HIRE, CURRENT_DATE
FROM EMPLOYEE_PAY_TBL;
```

6. 每名雇员是在星期几被雇用的？
7. 今天的儒略日期（积日）是多少？
8. 输入3行SQL代码。第1行获得系统时间（参考练习1），第2行将系统时间转换成日期型数据，第3行将系统时间转换成时间值。

## [第四部分 创建复杂的数据库查询](#)

第13章 在查询里结合表

第14章 使用子查询定义未确定数据

第15章 组合多个查询

### [第13章 在查询里结合表](#)

本章的重点包括：

简介表的结合

不同类型的结合

如何、何时使用结合

表结合的范例

不恰当表结合的影响

在查询中利用别名对表进行重命名

到目前为止，我们执行的数据库查询只是从一个表里获取数据。这一章将介绍如何在一个查询里结合多个表来获取数据。

#### [13.1 从多个表获取数据](#)

能够从多个表选择数据是SQL最强大的特性之一。如果没有这种能力，关系型数据库的整个概念就无法实现了。有时单表查询就可以得到有用的信息，但在现实世界里，最实用的查询是要从数据库里的多个表获取数据。

第4章已经介绍过，关系型数据库为达到简单和易于管理的目的，被分解为较小的、更易管理的表。正是由于表被分解为较小的表，它们通过共有字段（主键和外键）形成相互关联的表，并且能够通过这些字

段结合在一起。

有人就会问了，既然最终还是要利用重新结合表来获取需要的数据，那么为什么还要对表进行规格化呢？在实际应用中，我们很少会从表里选择全部数据，因此最好是根据每个查询的需求进行挑选。虽然数据库的规格化会对性能造成一点影响，但从整体来说，编程和维护都更加容易了。需要记住的是，规格化的主要目的是减少冗余和提高数据完整性。数据库管理员的最终目标是确保数据安全。

## [13.2 结合的类型](#)

结合是把两个或多个表组合在一起来获取数据。不同的实现具有多种结合表的方式，本章将介绍最常用的结合方式，它们是：

等值结合或内部结合；

非等值结合；

外部结合；

自结合。

### [13.2.1 结合条件的位置](#)

从前面的课程可以知道，SELECT和FROM是SQL语句的必要子句；而在结合表时，WHERE子句是必要的。要结合的表列在FROM子句里，而结合是在WHERE子句里完成的。多个操作符可以用于结合表，比如=、<、>、<>、<=、>=、!=、BETWEEN、LIKE和NOT，其中最常用的是等于号。

### [13.2.2 等值结合](#)

最常用也是最重要的结合就是等值结合，也被称为内部结合。等值结合利用通用字段结合两个表，而这个字段通常是每个表里的主键。

等值结合的语法如下所示：

```
SELECT TABLE1.COLUMN1, TABLE2.COLUMN2...  
FROM TABLE1, TABLE2 [, TABLE3 ]  
WHERE TABLE1.COLUMN_NAME = TABLE2.COLUMN_NAME  
[ AND TABLE1.COLUMN_NAME = TABLE3.COLUMN_NAME ]
```

具体范例如下：

```
SELECT EMPLOYEE_TBL.EMP_ID,  
       EMPLOYEE_PAY_TBL.DATE_HIRE  
FROM EMPLOYEE_TBL,  
     EMPLOYEE_PAY_TBL  
WHERE EMPLOYEE_TBL.EMP_ID = EMPLOYEE_PAY_TBL.EMP_ID;
```

这个SQL语句返回雇员标识和雇佣日期。雇员标识来自于表 **EMPLOYEE\_TBL**（虽然它存在于两个表里，但我们必须指定一个表），而雇佣日期来自于表 **EMPLOYEE\_PAY\_TBL**。由于雇员标识在两个表都存在，所以字段名称前面必须用表名加以修饰，从而让数据库服务程序明确到哪里获取数据。

注意：在**SQL**语句中使用缩排

注意到在上面这个范例SQL语句里使用了缩排方式来提高可读性。缩排方式不是必须的，但是推荐使用。

下面的范例从表**EMPLOYEE\_TBL**和**EMPLOYEE\_PAY\_TBL**里获取数据，使用了等值结合。

```

SELECT EMPLOYEE_TBL.EMP_ID, EMPLOYEE_TBL.LAST_NAME,
       EMPLOYEE_PAY_TBL.POSITION
FROM EMPLOYEE_TBL, EMPLOYEE_PAY_TBL
WHERE EMPLOYEE_TBL.EMP_ID = EMPLOYEE_PAY_TBL.EMP_ID;

```

```

EMP_ID    LAST_NAM POSITION
-----
311549902 STEPHENS MARKETING
442346889 PLEW      TEAM LEADER
213764555 GLASS     SALES MANAGER
313782439 GLASS     SALESMAN
220984332 WALLACE  SHIPPER
443679012 SPURGEON SHIPPER

```

6 rows selected.

SELECT子句里每个字段名称都以表名作为前缀，从而准确标识各个字段。在查询中，这被称为限定字段，它只有在字段存在于多个表时才有必要。在调试或修改SQL代码时，我们通常会对全部字段进行限定，从而提高一致性并减少问题。

另外，SQL里可以利用 INNER JOIN语法来提高可读性，如下所示：

```

SELECT TABLE1.COLUMN1, TABLE2.COLUMN2...
FROM TABLE1
INNER JOIN TABLE2 ON TABLE1.COLUMN_NAME = TABLE2.COLUMN_NAME

```

在这种方式里，WHERE 子句里的结合操作符被去掉了，取而代之的是关键字 INNER JOIN。要被结合的表位于JOIN之后，而结合操作符位于关键字ON之后。下面的范例使用JOIN语法来返回与前例一样的结果：

```

SELECT EMPLOYEE_TBL.EMP_ID,
       EMPLOYEE_PAY_TBL.DATE_HIRE
FROM EMPLOYEE_TBL
INNER JOIN EMPLOYEE_PAY_TBL
ON EMPLOYEE_TBL.EMP_ID = EMPLOYEE_PAY_TBL.EMP_ID;

```

上述两个范例的语法虽然不同，但它们都返回一样的结果。

### [13.2.3 使用表的别名](#)

使用表的别名意味着在SQL语句里对表进行重命名，这是一种临时性的改变，表在数据库里的实际名称不会受到影响。稍后我们就会看到，让表具有别名是完成自结合的必要条件。给表起别名一般是为了减少键盘输入，从而得到更短、更易读的SQL语句。另外，输入较少就意味着更少的输入错误。而且，在对别名进行引用时，由于它一般比较短，而且更能准确描述数据，所以编程错误也会更少。给表起别名同时也意味着被选择字段必须用表的别名加以修饰。下面是使用表的别名的一些范例：

```

SELECT E.EMP_ID, EP.SALARY, EP.DATE_HIRE, E.LAST_NAME
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL EP
WHERE E.EMP_ID = EP.EMP_ID
AND EP.SALARY > 20000;

```

这个 SQL 语句里给表设置了别名。EMPLOYEE\_TBL 被重命名为 E，EMPLOYEE\_PAY\_TBL 被重命名为 EP。选择什么名称作为别名没有限制，这里使用 E 是因为 EMPLOYEE\_TBL 以 E 开头。虽然 EMPLOYEE\_PAY\_TBL 也以 E 开头，但不能再使用 E 了，所以用第一个字母 E 和第二个单词的第一个字母 (P) 组成 EP 作为这个表的别名。被选择的字段由相应表的别名加以修饰。注意 WHERE 子句里使用的 SALARY 字段也必须用表的别名加以修饰。



#### 13.2.4 不等值结合

不等值结合根据同一个字段在两个表里值不相等来实现结合，其语法如下所示：

```
FROM TABLE1, TABLE2 [, TABLE3 ]  
WHERE TABLE1.COLUMN_NAME != TABLE2.COLUMN_NAME  
[ AND TABLE1.COLUMN_NAME != TABLE2.COLUMN_NAME ]
```

具体范例如下：

```
SELECT EMPLOYEE_TBL.EMP_ID, EMPLOYEE_PAY_TBL.DATE_HIRE  
FROM EMPLOYEE_TBL,  
     EMPLOYEE_PAY_TBL  
WHERE EMPLOYEE_TBL.EMP_ID != EMPLOYEE_PAY_TBL.EMP_ID;
```

下面的SQL语句返回在两个表里没有相应记录的全部雇员的标识及雇佣日期，使用的就是不等值结合：

```
SELECT E.EMP_ID, E.LAST_NAME, P.POSITION
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL P
WHERE E.EMP_ID <> P.EMP_ID;
```

EMP_ID	LAST_NAM	POSITION
442346889	PLEW	MARKETING
213764555	GLASS	MARKETING
313782439	GLASS	MARKETING
220984332	WALLACE	MARKETING
443679012	SPURGEON	MARKETING
311549902	STEPHENS	TEAM LEADER
213764555	GLASS	TEAM LEADER
313782439	GLASS	TEAM LEADER
220984332	WALLACE	TEAM LEADER
443679012	SPURGEON	TEAM LEADER
311549902	STEPHENS	SALES MANAGER
442346889	PLEW	SALES MANAGER
313782439	GLASS	SALES MANAGER
220984332	WALLACE	SALES MANAGER
443679012	SPURGEON	SALES MANAGER
311549902	STEPHENS	SALESMAN
442346889	PLEW	SALESMAN
213764555	GLASS	SALESMAN
220984332	WALLACE	SALESMAN
443679012	SPURGEON	SALESMAN
311549902	STEPHENS	SHIPPER
442346889	PLEW	SHIPPER
213764555	GLASS	SHIPPER
313782439	GLASS	SHIPPER
443679012	SPURGEON	SHIPPER

警告：不等值组合可能会产生多余数据

在使用不等值结合时，可能会得到很多无用的数据，其结果需要仔细检查。

每个表里只有 6 条记录，上面这个 SQL 语句为什么会返回 30 行记录呢？对于表EMPLOYEE\_TBL里的每条记录，在EMPLOYEE\_PAY\_TBL里都有一条相应的记录。由于在表结合时测试

的是不相等条件，所以第一个表里每条记录在与第二个表里的全部记录进行比较时，除其对应的记录，其他记录都满足条件。这意味着每条记录都与第二个表里5条不相关记录满足条件，因此6乘以5得到总共30条记录。

在前面小节中使用等值结合的例子中，第一个表里的每条记录都只与第二个表里的一行记录相匹配（其对应的记录），所以6乘以1得到总共6条记录。

### 13.2.5 外部结合

外部结合会返回一个表里的全部记录，即使对应的记录在第二个表里不存在。加号（+）用于在查询中表示外部结合，放在WHERE子句里表名的后面。具有加号的表是没有匹配记录的表。在很多实现里，外部结合被划分为左外部结合、右外部结合和全外部结合。

注意：结合的语法结构多变

关于外部结合的使用与语法请查看具体实现的文档。很多主流实现都使用“+”表示外部结合，但这并不是标准。实际上，相同数据库实现的不同版本，其相关规定也不尽相同。例如，Microsoft SQL Server 2000支持这种语法，但其2005及以上版本却不支持。所以，在使用这种语法结构时务必要小心。

外部结合的一般语法如下所示：

```
FROM TABLE1
{RIGHT | LEFT | FULL} [OUTER] JOIN
ON TABLE2
```

Oracle的语法是：

```
FROM TABLE1, TABLE2 [, TABLE3 ]
WHERE TABLE1.COLUMN_NAME[(+)] = TABLE2.COLUMN_NAME[(+)]
[ AND TABLE1.COLUMN_NAME[(+)] = TABLE3.COLUMN_NAME[(+)] ]
```

注意：外部结合的应用

外部结合只能用于JOIN条件的一侧，但可以在JOIN条件里对同一个表里的多个字段进行外部结合。

外部结合的概念将在下面的两个范例里加以解释。第一个范例选择了产品描述和订购数量，这两个值取自两个单独的表里。需要注意的是，并不是每件产品在表ORDERS\_TBL里都有相应的记录。这里执行了一个普通的等值结合：

```
SELECT P.PROD_DESC, O.QTY
FROM PRODUCTS_TBL P,
      ORDERS_TBL O
WHERE P.PROD_ID = O.PROD_ID;
```

PROD_DESC	QTY
PLASTIC PUMPKIN 18 INCH	2
LIGHTED LANTERNS	10
PLASTIC SPIDERS	30
LIGHTED LANTERNS	20
FALSE PARAFFIN TEETH	20
PUMPKIN CANDY	10
FALSE PARAFFIN TEETH	10
WITCH COSTUME	5
CANDY CORN	45
LIGHTED LANTERNS	25
PLASTIC PUMPKIN 18 INCH	25
WITCH COSTUME	30
FALSE PARAFFIN TEETH	15
PLASTIC SPIDERS	50
PLASTIC PUMPKIN 18 INCH	25
PLASTIC PUMPKIN 18 INCH	25
WITCH COSTUME	1

17 rows selected.

这里只得到了7种产品的17条记录，但产品共有9种。我们想显示全

部的产品，不管它是否有订单。

下面的范例通过使用外部结合来达到我们的目的，这里使用的是 Oracle 语法：

```
SELECT P.PROD_DESC, O.QTY
FROM PRODUCTS_TBL P,
      ORDERS_TBL O
WHERE P.PROD_ID = O.PROD_ID(+);
```

PROD_DESC	QTY
-----	-----
WITCH COSTUME	5
WITCH COSTUME	30
WITCH COSTUME	1
ASSORTED MASKS	NULL
FALSE PARAFFIN TEETH	20
FALSE PARAFFIN TEETH	10
FALSE PARAFFIN TEETH	15
ASSORTED COSTUMES	NULL
PLASTIC PUMPKIN 18 INCH	2
PLASTIC PUMPKIN 18 INCH	25
PLASTIC PUMPKIN 18 INCH	25
PLASTIC PUMPKIN 18 INCH	25
PUMPKIN CANDY	10
PLASTIC SPIDERS	30
PLASTIC SPIDERS	50
CANDY CORN	45
LIGHTED LANTERNS	10
LIGHTED LANTERNS	20
LIGHTED LANTERNS	25

19 rows selected.

这里也可以使用前面介绍的比较繁琐的语法结构，获得相同的结果。下面的范例就使用了这种繁琐结构，但清晰易懂。

```

SELECT P.PROD_DESC, O.QTY
FROM PRODUCTS_TBL P
LEFT OUTER JOIN  ORDERS_TBL O
ON P.PROD_ID = O.PROD_ID;

```

PROD_DESC	QTY
-----	-----
WITCH COSTUME	5
WITCH COSTUME	30
WITCH COSTUME	1
ASSORTED MASKS	NULL
FALSE PARAFFIN TEETH	20
FALSE PARAFFIN TEETH	10
FALSE PARAFFIN TEETH	15
ASSORTED COSTUMES	NULL
PLASTIC PUMPKIN 18 INCH	2
PLASTIC PUMPKIN 18 INCH	25
PLASTIC PUMPKIN 18 INCH	25
PLASTIC PUMPKIN 18 INCH	25
PUMPKIN CANDY	10
PLASTIC SPIDERS	30
PLASTIC SPIDERS	50
CANDY CORN	45
LIGHTED LANTERNS	10
LIGHTED LANTERNS	20
LIGHTED LANTERNS	25

19 rows selected.

这个查询返回了全部的产品，不论它是否有相应的订单。外部结合会包含表PRODUCT\_TBL里的全部记录，不管它在表ORDER\_TBL里是否有对应的记录。

### [13.2.6 自结合](#)

自结合利用表别名在SQL语句对表进行重命名，像处理两个表一样把表结合到自身。其语法如下所示：

```
SELECT A.COLUMN_NAME, B.COLUMN_NAME, [ C.COLUMN_NAME ]
FROM TABLE1 A, TABLE2 B [ , TABLE3 C ]
WHERE A.COLUMN_NAME = B.COLUMN_NAME
[ AND A.COLUMN_NAME = C.COLUMN_NAME ]
```

具体范例如下：

```
SELECT A.LAST_NAME, B.LAST_NAME, A.FIRST_NAME
FROM EMPLOYEE_TBL A,
     EMPLOYEE_TBL B
WHERE A.LAST_NAME = B.LAST_NAME;
```

这个SQL语句返回表EMPLOYEE\_TBL里所有姓相同的雇员的姓名。当需要的数据都位于同一个表里，而我们又必须对记录进行一些比较时，就可以使用自结合。

还可以像下面这样利用 INNER JOIN来得到同样的结果：

```
SELECT A.LAST_NAME, B.LAST_NAME, A.FIRST_NAME
FROM EMPLOYEE_TBL A
INNER JOIN EMPLOYEE_TBL B
ON A.LAST_NAME = B.LAST_NAME;
```

使用自结合的另一个常见范例是：假设有一个表保存了雇员标识码、姓名、雇员主管的标识码。我们想列出所有雇员及其主管的姓名，问题在于雇员主管的姓名并不是表里的一个字段：

```
SELECT * FROM EMP;
```

ID	NAME	MGR_ID
1	JOHN	0
2	MARY	1
3	STEVE	1
4	JACK	2
5	SUE	2

在下面的语句里，我们在FROM子句里包含了表EMP两次，让表具有两个别名。这样我们就可以像使用两个不同的表一样进行操作。所有的主管也都是雇员，所以 JOIN 条件比较第一个表里的雇员标识号码与第二个表里的主管标识号码。第一个表就像是保存雇员信息的表，而第二个表就像是保存主管信息的表：

```
SELECT E1.NAME, E2.NAME  
FROM EMP E1, EMP E2  
WHERE E1.MGR_ID = E2.ID;
```

NAME	NAME
-----	-----
MARY	JOHN
STEVE	JOHN
JACK	MARY
SUE	MARY

### [13.2.7 结合多个主键](#)

大多数结合操作都会基于一个表里的主键和另一个表里的主键来合并数据。根据数据库的设计情况，有时我们需要结合多个主键来描述数据库里的数据。比如可能某个表的主键由多个字段组成，可能某个表的外键由多个字段组成，分别引用多个主键。

比如下面这个Oracle表：



```

SQL> desc prod
Name                                     Null?      Type
-----
SERIAL_NUMBER                           NOT NULL   NUMBER(10)
VENDOR_NUMBER                           NOT NULL   NUMBER(10)
PRODUCT_NAME                            NOT NULL   VARCHAR2(30)
COST                                     NOT NULL   NUMBER(8,2)

SQL> desc ord
Name                                     Null?      Type
-----
ORD_NO                                  NOT NULL   NUMBER(10)
PROD_NUMBER                            NOT NULL   NUMBER(10)
VENDOR_NUMBER                           NOT NULL   NUMBER(10)
QUANTITY                                NOT NULL   NUMBER(5)
ORD_DATE                                NOT NULL   DATE

```

PROD里的主键是由字段SERIAL\_NUMBER和VENDOR\_NUMBER组成的。也许两个产品在配送公司具有相同的序列号，但在每个商家的序列号都是唯一的。

ORD里的外键也是由字段SERIAL\_NUMBER和VENDOR\_NUMBER组成的。

在从两个表（PROD和ORD）里选择数据时，结合操作可能是这样的：

```

SELECT P.PRODUCT_NAME, O.ORD_DATE, O.QUANTITY
FROM PROD P, ORD O
WHERE P.SERIAL_NUMBER = O.SERIAL_NUMBER
      AND P.VENDOR_NUMBER = O.VENDOR_NUMBER;

```

类似地，如果要使用 INNER JOIN，只需要在关键字ON之后列出多个结合操作：

```

SELECT P.PRODUCT_NAME, O.ORD_DATE, O.QUANTITY
FROM PROD P,
INNER JOIN ORD O ON P.SERIAL_NUMBER = O.SERIAL_NUMBER
      AND P.VENDOR_NUMBER = O.VENDOR_NUMBER;

```

### [13.3 需要考虑的事项](#)

在使用结合之前需要考虑一些事情：基于什么字段进行结合、是否有公用字段进行结合、性能问题。查询里的结合越多，数据库需要完成的工作就越多，也就意味着需要越多的时间来获取数据。在从规格化的数据库里获取数据时，结合是不可避免的，但需要从逻辑角度来确定结合是正确执行的。不恰当的结合会导致严重的性能下降和不准确的查询结果。关于性能的问题将在第18章详细介绍。

#### [13.3.1 使用基表](#)

要结合什么？如果需要从两个表里获取数据，但它们又没有公用字段，我们就必须结合另一个表，这个表与前两个表都有公用字段，这个表就被称为基表。基表用于结合具有公用字段的一个或多个表，或是结合没有公用字段的多个表。下面是基表范例要用到的表：

CUSTOMER_TBL			
CUST_ID	VARCHAR(10)	NOT NULL	primary key
CUST_NAME	VARCHAR(30)	NOT NULL	
CUST_ADDRESS	VARCHAR(20)	NOT NULL	
CUST_CITY	VARCHAR(15)	NOT NULL	
CUST_STATE	VARCHAR(2)	NOT NULL	
CUST_ZIP	INTEGER(5)	NOT NULL	
CUST_PHONE	INTEGER(10)		
CUST_FAX	INTEGER(10)		
ORDERS_TBL			
ORD_NUM	VARCHAR(10)	NOT NULL	primary key
CUST_ID	VARCHAR(10)	NOT NULL	
PROD_ID	VARCHAR(10)	NOT NULL	
QTY	INTEGER(6)	NOT NULL	
ORD_DATE	DATETIME		
PRODUCTS_TBL			
PROD_ID	VARCHAR(10)	NOT NULL	primary key
PROD_DESC	VARCHAR(40)	NOT NULL	
COST	DECIMAL(6,2)	NOT NULL	

假设我们要使用表CUSTOMERS\_TBL和PRODUCTS\_TBL，但它们之间没有公用字段。现在来看表ORDERS\_TBL，它与表CUSTOMER\_TBL 可以通过 CUST\_ID字段结合，与表PRODUCTS\_TBL 可以通过PROD\_ID字段结合。相应的JOIN条件及结果如下所示：

```

SELECT C.CUST_NAME, P.PROD_DESC
FROM CUSTOMER_TBL C,
     PRODUCTS_TBL P,
     ORDERS_TBL O
WHERE C.CUST_ID = O.CUST_ID
     AND P.PROD_ID = O.PROD_ID;

```

CUST_NAME	PROD_DESC
LESLIE GLEASON	WITCH COSTUME
SCHYLERS NOVELTIES	PLASTIC PUMPKIN 18 INCH
WENDY WOLF	PLASTIC PUMPKIN 18 INCH
GAVINS PLACE	LIGHTED LANTERNS
SCOTTYS MARKET	FALSE PARAFFIN TEETH
ANDYS CANDIES	KEY CHAIN

6 rows selected.

注意：别名的使用

注意WHERE子句里的表别名和它们如何用于字段。

### [13.3.2 笛卡尔积](#)

笛卡尔积是笛卡尔结合或“无结合”的结果。如果从两个或多个没有结合的表里获取数据，输出结果就是所有被选表里的全部记录。如果表的规模很大，其结果可能是几十万，甚至是数百万行数据。因此，在从两个或多个表里获取数据时，强烈建议使用WHERE子句。笛卡尔积通常也被称为交叉结合。

其语法如下所示：

```

FROM TABLE1, TABLE2 [, TABLE3 ]
WHERE TABLE1, TABLE2 [, TABLE3 ]

```

下面是交叉结合（或称为可怕的笛卡尔积）的范例：

```

SELECT E.EMP_ID, E.LAST_NAME, P.POSITION
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL P;

```

EMP_ID	LAST_NAM	POSITION
311549902	STEPHENS	MARKETING
442346889	PLEW	MARKETING
213764555	GLASS	MARKETING
313782439	GLASS	MARKETING
220984332	WALLACE	MARKETING
443679012	SPURGEON	MARKETING
311549902	STEPHENS	TEAM LEADER
442346889	PLEW	TEAM LEADER
213764555	GLASS	TEAM LEADER
313782439	GLASS	TEAM LEADER
220984332	WALLACE	TEAM LEADER
443679012	SPURGEON	TEAM LEADER
311549902	STEPHENS	SALES MANAGER
442346889	PLEW	SALES MANAGER
213764555	GLASS	SALES MANAGER
313782439	GLASS	SALES MANAGER
220984332	WALLACE	SALES MANAGER
443679012	SPURGEON	SALES MANAGER
311549902	STEPHENS	SALESMAN
442346889	PLEW	SALESMAN
213764555	GLASS	SALESMAN
313782439	GLASS	SALESMAN
220984332	WALLACE	SALESMAN
443679012	SPURGEON	SALESMAN
311549902	STEPHENS	SHIPPER
442346889	PLEW	SHIPPER
213764555	GLASS	SHIPPER
313782439	GLASS	SHIPPER
220984332	WALLACE	SHIPPER
443679012	SPURGEON	SHIPPER
311549902	STEPHENS	SHIPPER
442346889	PLEW	SHIPPER
213764555	GLASS	SHIPPER
313782439	GLASS	SHIPPER
220984332	WALLACE	SHIPPER
443679012	SPURGEON	SHIPPER

36 rows selected.

虽然没有执行 JOIN 操作，数据还是取自两个单独的表。由于我们没有指定第一个表里的记录如何与第二个表里的记录相结合，数据库服务程序把第一个表里每行记录都与第二个表里的全部记录相匹配。每个表都有6条记录，所以最终结果是6乘以6共计36条记录。

为了更好地理解笛卡尔积是如何得到的，再看下面这个范例：

```
SQL> SELECT X FROM TABLE1;
```

```
X  
-  
A  
B  
C  
D
```

```
4 rows selected.
```

```
SQL> SELECT V FROM TABLE2;
```

```
X  
-  
A  
B  
C  
D
```

```
4 rows selected.
```

```
SQL> SELECT TABLE1.X, TABLE2.X  
2* FROM TABLE1, TABLE2;
```

```
X X  
- -  
A A  
B A  
C A  
D A  
A B  
B B  
C B  
D B  
A C  
B C  
C C  
D C  
A D  
B D  
C D  
D D
```

```
16 rows selected.
```

警告：务必确保所有的表都结合完毕

在查询里结合多个表要特别小心。如果查询里的两个表没有结合，

而且每个表都包含 1 000 行数据，那么笛卡尔积就会是 1 000 乘以 1 000，也就是1 000 000 行数据。在处理大量数据时，笛卡尔积有时会导致主机停止或崩溃。因此，对于DBA和系统管理员来说，密切监视长时间运行的查询是件很重要的工作。

## 13.4 小结

本章介绍了SQL最强大的功能之一：表的结合。想象一下，如果在查询里只能从一个表获取数据，那我们将受到多么大的局限。这里介绍了多种结合类型，它们分别具有自己的功能。内部结合可以根据相等或不相等的条件连接多个表里的数据。外部结合是相当强大的，即使在被结合的表没有匹配数据时，也能从中获取数据。自结合用于把表与自身相结合。对于交叉结合，也就是笛卡尔积，要特别小心，它是多个表没有进行任何结合的结果，经常会产生大量不必要的结果。因此，在从多个表里获取数据时，一定要根据相关联的字段（通常是主键）把表进行结合。如果没有恰当地对表进行结合，可能会产生不完整或不正确的输出结果。

## 13.5 问与答

问：在结合表时，它们的结合次序必须与它们在**FROM**子句里出现的次序一样吗？

答：不必，它们不必以同样的次序出现。但是，表在**FROM**里的次序和表被结合的次序可能会对性能有所影响。

问：在使用基表结合没有关联的表时，必须从基表里选择字段吗？

答：不必。使用基表结合不相关的表并不要求从基表里选择字段。

问：在结合表时可以基于多个字段吗？



答：可以。有些查询要求基于多个字段进行结合，才能描述表的记录之间的完整关系。

## 13.6 实践

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### 13.6.1 测验

1. 如果不论相关表里是否存在匹配的记录，都要从表里返回记录，应该使用什么类型的结合？
2. JOIN条件位于SQL语句的什么位置？
3. 使用什么类型的结合来判断相关表的记录之间的相等关系？
4. 如果从两个不同的表获取数据，但它们没有结合，会产生什么结果？
5. 使用如下的表：

ORDERS_TBL			
ORD_NUM	VARCHAR(10)	NOT NULL	primary key
CUST_ID	VARCHAR(10)	NOT NULL	
PROD_ID	VARCHAR(10)	NOT NULL	
QTY	Integer(6)	NOT NULL	
ORD_DATE	DATETIME		
PRODUCTS_TBL			
PROD_ID	VARCHAR(10)	NOT NULL	primary key
PROD_DESC	VARCHAR(40)	NOT NULL	
COST	DECIMAL(,2)	NOT NULL	

下面使用外部结合的语法正确吗？

如果使用繁琐语法，上述查询语句会是什么样子？

```
SELECT C.CUST_ID, C.CUST_NAME, O.ORD_NUM
FROM CUSTOMER_TBL C, ORDERS_TBL O
WHERE C.CUST_ID(+) = O.CUST_ID(+)
```

### [13.6.2 练习](#)

1. 在数据库中输入以下代码，研究得到的结果（笛卡尔积）：

```
SELECT E.LAST_NAME, E.FIRST_NAME, EP.DATE_HIRE
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL EP;
```

2. 输入以下命令来结合表EMPLOYEE\_TBL和EMPLOYEE\_PAY\_TBL：

```
SELECT E.LAST_NAME, E.FIRST_NAME, EP.DATE_HIRE
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL EP
WHERE E.EMP_ID = EP.EMP_ID;
```

3. 改写练习2里的SQL查询语句，使用 INNER JOIN语法。

4. 编写一个 SQL 语句，从表 EMPLOYEE\_TBL 返回 EMP\_ID、LAST\_NAME 和FIRST\_NAME字段，从表EMPLOYEE\_PAY\_TBL返回SALARY和BONUS字段。使用两种类型的 INNER JOIN技术。完成上述查询以后，再进一步计算出每个城市雇员的平均薪水是多少。

5. 尝试自己编写几条使用结合操作的查询语句。

## [第14章 使用子查询定义未确定数据](#)

本章的重点包括：

什么是子查询

使用子查询的原因

常规数据库查询中使用子查询的范例

子查询与数据操作命令

嵌入式子查询

本章介绍子查询的相关内容。子查询可以帮助用户更便捷地完成复杂的查询操作。

## [14.1 什么是子查询](#)

子查询也被称为嵌套查询，是位于另一个查询的 **WHERE** 子句里的查询，它返回的数据通常在主查询里作为一个条件，从而进一步限制数据库返回的数据。它可以用于 **SELECT**、**INSERT**、**UPDATE**和**DELETE**语句。

在某些情况下，子查询能够间接地基于一个或多个条件把多个表里的数据关联起来，从而代替结合操作。当查询里使用子查询时，子查询首先被执行，然后主查询再根据子查询返回的结果执行。子查询的结果用于在主查询的 **WHERE** 子句里处理表达式。子查询可以用于主查询的 **WHERE**子句或**HAVING**子句。逻辑和关系操作符，比如=、>、<、<>、!=、**IN**、**NOT IN**、**AND**、**OR**，可以用于子查询里，也可以在 **WHERE**或**HAVING**子句里对子查询进行操作。

注意：子查询规则

标准查询的规则同样也适用于子查询，结合操作、函数、转换和其他选项都可以在子查询里使用。

注意：使用缩进来提高可读性

注意范例中所使用的缩进。使用缩进基本上就是为了提高可读性。我们发现在查找**SQL**语句里的错误时，语句越整洁，就越容易阅读并发

现语法中的错误。

子查询必须遵循以下规则。

子查询必须位于圆括号里。

除非主查询里有多个字段让子查询进行比较，否则子查询的SELECT子句里只能有一个字段。

子查询里不能使用ORDER BY子句。在子查询里，我们可以利用GROUP BY子句实现ORDER BY功能。

返回多条记录的子查询只能与多值操作符（比如IN）配合使用。

SELECT列表里不能引用任何BLOB、ARRAY、CLOB或NCLOB类型的值。

子查询不能被包围在函数里。

操作符BETWEEN不能用于子查询，但子查询内部可以使用它。子查询的基本语法如下所示：

```
SELECT COLUMN_NAME
FROM TABLE
WHERE COLUMN_NAME = (SELECT COLUMN_NAME
                      FROM TABLE
                      WHERE CONDITIONS);
```

下面的范例展示了操作符 BETWEEN 与子查询的关系。首先是在子查询里使用BETWEEN的正确范例。

```
SELECT COLUMN_NAME
FROM TABLE_A
WHERE COLUMN_NAME OPERATOR (SELECT COLUMN_NAME
                             FROM TABLE_B)
                             WHERE VALUE BETWEEN VALUE)
```

不能够在子查询外使用BETWEEN。下面是错误地把BETWEEN用于子查询的范例：

```
SELECT COLUMN_NAME
FROM TABLE_A
WHERE COLUMN_NAME BETWEEN VALUE AND (SELECT COLUMN_NAME
                                       FROM TABLE_B)
```

### [14.1.1 子查询与SELECT语句](#)

虽然子查询也可以用于数据操作语句，但它最主要还是用于SELECT语句里，获取数据给主查询使用。

基本语法如下所示：

```
SELECT COLUMN_NAME [, COLUMN_NAME ]
FROM TABLE1 [, TABLE2 ]
WHERE COLUMN_NAME OPERATOR
      (SELECT COLUMN_NAME [, COLUMN_NAME ]
       FROM TABLE1 [, TABLE2 ]
       [ WHERE ])
```

下面是一个范例：

```
SELECT E.EMP_ID, E.LAST_NAME, E.FIRST_NAME, EP.PAY_RATE
FROM EMPLOYEE_TBL E, EMPLOYEE_PAY_TBL EP
WHERE E.EMP_ID = EP.EMP_ID
AND EP.PAY_RATE < (SELECT PAY_RATE
                   FROM EMPLOYEE_PAY_TBL
                   WHERE EMP_ID = '443679012');
```

上面这条SQL语句返回小时工资低于雇员443679012的所有雇员的标识、姓、名和小时工资。这时，我们不必准确知道（或关心）这个特定雇员的小时工资是多少，只想知道比这个雇员工资低的人都是谁。

注意：使用子查询来查找不确定的值

在不能确定条件里的准确数值时，通常可以使用子查询来实现。雇员220984332的薪水是不确定的，但子查询可以帮我们完成这些跑腿的工作。

下面的查询选择某个雇员的小时工资，这个查询将作为后面范例里的一个子查询。

```
SELECT PAY_RATE
FROM EMPLOYEE_PAY_TBL
WHERE EMP_ID = '220984332';
```

```
    PAY_RATE
-----
11
```

1 row selected.

前面的查询在下面查询的WHERE子句里充当一个子查询：

```
SELECT E.EMP_ID, E.LAST_NAME, E.FIRST_NAME, EP.PAY_RATE
FROM EMPLOYEE_TBL E, EMPLOYEE_PAY_TBL EP
WHERE E.EMP_ID = EP.EMP_ID
      AND EP.PAY_RATE > (SELECT PAY_RATE
                          FROM EMPLOYEE_PAY_TBL
                          WHERE EMP_ID = '220984332');
```

EMP_ID	LAST_NAME	FIRST_NAME	PAY_RATE
442346889	PLEW	LINDA	14.75
443679012	SPURGEON	TIFFANY	15

2 rows selected.

子查询的结果是11（见前一个范例），所以上面这个WHERE子句的条件实际上是：

```
AND EP.PAY_RATE > 11
```

在执行这个查询时，我们不知道特定雇员的小时工资是多少，但主查询还是可以把每个雇员的小时工资与子查询的结果进行比较。

### [14.1.2 子查询与INSERT语句](#)

子查询可以与数据操作语言（DML）配合使用。首先是INSERT语句，它将子查询返回的结果插入到另一个表。我们可以用字符函数、日期函数或数值函数对子查询里选择的数据进行调整。

注意：提交执行**DML**命令

在使用像 INSERT 语句这样的 DML 命令时，要记得使用 COMMIT 和ROLLBACK命令。

基本语法如下所示：

```
INSERT INTO TABLE_NAME [ (COLUMN1 [, COLUMN2 ]) ]  
SELECT [ *|COLUMN1 [, COLUMN2 ]  
FROM TABLE1 [, TABLE2 ]  
[ WHERE VALUE OPERATOR ]
```

下面是在INSERT语句里使用子查询的范例：

```
INSERT INTO RICH_EMPLOYEES  
SELECT E.EMP_ID, E.LAST_NAME, E.FIRST_NAME, EP.PAY_RATE  
FROM EMPLOYEE_TBL E, EMPLOYEE_PAY_TBL EP  
WHERE E.EMP_ID = EP.EMP_ID  
      AND EP.PAY_RATE > (SELECT PAY_RATE  
                          FROM EMPLOYEE_PAY_TBL  
                          WHERE EMP_ID = '220984332');  
  
2 rows created.
```

这个 INSERT 语句把小时工资高于雇员 220984332 的所有雇员的 EMP\_ID、LAST\_NAME、FIRST\_NAME和PAY\_RATE插入到一个名为 RICH\_EMPLOYEES的表里。

### [14.1.3 子查询与UPDATE语句](#)

子查询可以与UPDATE语句配合使用来更新一个表里的一个或多个字段，其基本语法如下所示：

```

UPDATE TABLE
SET COLUMN_NAME [ , COLUMN_NAME ) ] =
    (SELECT ]COLUMN_NAME [ , COLUMN_NAME ) ]
FROM TABLE
[ WHERE ]

```

下面的范例展示了如何在 UPDATE 语句里使用子查询。第一个查询返回居住在Indianapolis的全部雇员的标识，可以看到共有4人满足条件。

```

SELECT EMP_ID
FROM EMPLOYEE_TBL
WHERE CITY = 'INDIANAPOLIS';

```

```

EMP_ID
-----
442346889
313782439
220984332
443679012

```

```

4 rows selected.

```

前面这个查询作为一个子查询用于下面这个UPDATE语句里。前面的结果说明了子查询会返回的雇员数量。下面是使用这个子查询的UPDATE语句：

```

UPDATE EMPLOYEE_PAY_TBL
SET PAY_RATE = PAY_RATE * 1.1
WHERE EMP_ID IN (SELECT EMP_ID
                  FROM EMPLOYEE_TBL
                  WHERE CITY = 'INDIANAPOLIS');

```

```

4 rows updated.

```

不出所料，有4条记录被更新了。与前一小节的子查询范例不同的是，这个子查询返回多条记录，因此要使用操作符IN而不是等号（IN可



以把一个表达式与列表里的多个值进行比较)。这里如果使用了等号，数据库会返回一个错误消息。

#### [14.1.4 子查询与DELETE语句](#)

子查询也可以与DELETE语句配合使用，其基本语法如下所示：

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
                (SELECT COLUMN_NAME
                  FROM TABLE_NAME)
[ WHERE) ]
```

下面的范例从表EMPLOYEE\_PAY\_TBL里删除GRANDON GLASS的记录。这时我们不知道Brandon的标识号码，但可以利用一个子查询，根据FIRST\_NAME和LAST\_NAME字段的值从表EMPLOYEE\_TBL里获取他的标识号码。

```
DELETE FROM EMPLOYEE_PAY_TBL
WHERE EMP_ID = (SELECT EMP_ID
                FROM EMPLOYEE_TBL
                WHERE LAST_NAME = 'GLASS'
                  AND FIRST_NAME = 'BRANDON');

1 row deleted.
```

#### [14.2 嵌套的子查询](#)

子查询可以嵌入到另一个子查询里，就像子查询嵌套在普通查询里一样。在有子查询时，子查询先于主查询执行。类似地，在嵌套的子查询里，最内层的子查询先被执行，然后再依次执行外层的子查询，直到主查询。

注意：确认实现对子查询的限制规定

一个语句里能够嵌套的子查询的数量取决于具体的实现，请查看相应的文档。

嵌套子查询的基本语法如下所示：

```
SELECT COLUMN_NAME [, COLUMN_NAME ]
FROM TABLE1 [, TABLE2 ]
WHERE COLUMN_NAME OPERATOR (SELECT COLUMN_NAME
                             FROM TABLE
                             WHERE COLUMN_NAME OPERATOR
                             (SELECT COLUMN_NAME
                              FROM TABLE
                              [ WHERE COLUMN_NAME OPERATOR VALUE ]))
```

下面的范例使用了两个子查询，一个嵌套在另一个之内。这个范例返回一些顾客的信息，这些顾客的订单的数量乘以单个订单的结果大于全部产品的价格总和。

```
SELECT CUST_ID, CUST_NAME
FROM CUSTOMER_TBL
WHERE CUST_ID IN (SELECT O.CUST_ID
                  FROM ORDERS_TBL O, PRODUCTS_TBL P
                  WHERE O.PROD_ID = P.PROD_ID
                  AND O.QTY * P.COST > (SELECT SUM(COST)
                                         FROM
                                         PRODUCTS_TBL));
```

CUST_ID	CUST_NAME
090	WENDY WOLF
232	LESLIE GLEASON
287	GAVINS PLACE
43	SCHYLERS NOVELTIES
432	SCOTTYS MARKET
560	ANDYS CANDIES

6 rows selected.

---

警告：使用**WHERE**子句

不要忘记在UPDATE和DELETE语句里使用WHERE子句，否则目标表里的全部数据都会被更新或删除。可以先使用一个带有 WHERE 子

句的SELECT语句进行查询，以便确认所要操作的数据准确无误。详情请见第5章的内容。

共有6条记录满足两个子查询的条件。

下面分别是两个子查询的结果，可以帮助我们更好地理解主查询是如何运行的。

```
SELECT SUM(COST) FROM PRODUCTS_TBL;
```

```
SUM(COST)
-----
138.08
```

```
1 row selected.
```

```
SELECT O.CUST_ID
FROM ORDERS_TBL O, PRODUCTS_TBL P
WHERE O.PROD_ID = P.PROD_ID
      AND O.QTY + P.COST > 138.08;
```

```
CUST_ID
-----
43
287
```

```
2 rows selected.
```

当最内层子查询执行完成之后，主查询实际上就变成这样：

```
SELECT CUST_ID, CUST_NAME
FROM CUSTOMER_TBL
WHERE CUST_ID IN (SELECT O.CUST_ID
                  FROM ORDERS_TBL O, PRODUCTS_TBL P
                  WHERE O.PROD_ID = P.PROD_ID
                        AND O.QTY + P.COST > 138.08);
```

当外层子查询也执行完成之后，主查询就是这样了：

```
SELECT CUST_ID, CUST_NAME
FROM CUSTOMER_TBL
WHERE CUST_ID IN (287,43);
```

下面是最终的结果：

CUST_ID	CUST_NAME
43	SCHYLERS NOVELTIES
287	GAVINS PLACE

2 rows selected.

警告：多个子查询可能会产生问题

使用多个子查询可能会延长响应时间，还可能降低结果的准确性，因为代码里可能存在错误。

### [14.3 关联子查询](#)

关联子查询在很多SQL实现里都存在，它的概念属于ANSI标准。关联子查询是依赖主查询里的信息的子查询。这意味着子查询里的表可以与主查询里的表相关联。

在下面这个范例里，子查询里结合的表CUSTOMER\_TBL和ORDERS\_TBL依赖于主查询里CUSTOMER\_TBL的别名（C）。这个查询返回订购超过10件物品的顾客的姓名。

```

SELECT C.CUST_NAME
FROM CUSTOMER_TBL C
WHERE 10 < (SELECT SUM(O.QTY)
            FROM ORDERS_TBL O
            WHERE O.CUST_ID = C.CUST_ID);

```

```

CUST_NAME
-----

```

```

SCOTTYS MARKET
SCHYLLERS NOVELTIES
MARYS GIFT SHOP

```

3 rows selected.

下面这个语句对子查询进行了一点修改，显示每个顾客订购的物品数量。

```

SELECT C.CUST_NAME, SUM(O.QTY)
FROM CUSTOMER_TBL C,
     ORDERS_TBL O
WHERE C.CUST_ID = O.CUST_ID
GROUP BY C.CUST_NAME;

```

CUST_NAME	SUM(O.QTY)
-----	-----
ANDYS CANDIES	1
GAVINS PLACE	10
LESLIE GLEASON	1
MARYS GIFT SHOP	100
SCHYLLERS NOVELTIES	25
SCOTTYS MARKET	20
WENDY WOLF	2

7 rows selected.

在这个范例里，GROUP BY子句是必需的，因为另一个字段被汇总函数SUM使用了。这样我们就得到了每个顾客订购的数量总和。在前一个子查询里，SUM函数用于获得整个查询的总和，就不是必须使用

GROUP BY子句了。

## [14.4 子查询的效率](#)

子查询会对执行效率产生影响。在应用子查询前，必须首先考虑好其所带来的影响。由于子查询会在主查询之前进行，所以子查询所花费的时间，会直接影响整个查询所需要的时间。看下面的范例。

注意：适当使用关联子查询

在进行关联子查询时，如果要在子查询中使用某个表，必须首先在主查询中引用这个表。

```
SELECT CUST_ID, CUST_NAME
FROM CUSTOMER_TBL
WHERE CUST_ID IN (SELECT O.CUST_ID
                  FROM ORDERS_TBL O, PRODUCTS_TBL P
                  WHERE O.PROD_ID = P.PROD_ID
                  AND O.QTY + P.COST < (SELECT SUM(COST)
                                         FROM
                                         PRODUCTS_TBL));
```

如果PRODUCTS\_TBL表中包含有数以千计的产品信息，而ORDERS\_TBL表中则保存了数以百万计的订单信息，想象一下这将意味着什么。对PRODUCTS\_TBL表进行汇总，并与ORDERS\_TBL进行关联，将在很大程度上影响操作的运行速度。所以，在需要使用子查询从数据库中获得相应信息的时候，务必考虑清楚子查询的执行效率。

## [14.5 小结](#)

简单来说，子查询就是在另一个查询里执行的查询，用于进一步设置查询的条件。子查询可以用于SQL语句的WHERE子句或HAVING子句。它不仅可以在查询里使用，还可以用于DML（数据操作语言）语

句，比如INSERT、UPDATE和DELETE，但这时要注意遵守DML的基本规则。

子查询的语法实质上与普通查询是一样的，只是有一些细微的限制。其中之一是不能使用ORDER BY子句，但可以使用GROUP BY子句，也能得到同样的效果。子查询可以向查询提供不必事先确定的条件，增强了SQL的功能灵活性。

## [14.6 问与答](#)

问：在子查询的范例里有很多的缩进，这是语法要求的吗？

答：当然不是，缩进只是把语句划分为多个部分，让语句更易于阅读和理解。

问：一个查询里能够嵌套的子查询数量是否有限制？

答：像允许嵌套的子查询数量、查询里能够结合的表的数量等限制都是取决于具体实现的。有些实现可能没有限制，但子查询嵌套太多可能会明显降低语句的性能。大多数限制受到实际的硬件、CPU速度和可用系统内存的影响，当然还有其他一些考虑。

问：调试具有子查询，特别是嵌套子查询的语句似乎很容易让人迷惑，有什么好方法来调试具有子查询的语句吗？

答：调试具有子查询的语句的最好方法是分几个部分对查询进行求值。首先，运算最内层的子查询，然后逐步扩展到主查询（这与数据库执行查询的次序一样）。在单独运行了每个子查询之后，就可以把子查询的返回值代入到主查询，检查主查询的逻辑是否正确。子查询带来的错误经常是由对其使用的操作符造成的，比如=、IN、<、>等。

## [14.7 实践](#)

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在

于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### 14.7.1 测验

1. 在用于SELECT语句时，子查询的功能是什么？
2. 在子查询与UPDATE语句配合使用时，能够更新多个字段吗？
3. 下面的语法正确吗？如果不正确，正确的语法应该是怎样？

**a.**

```
SELECT CUST_ID, CUST_NAME
      FROM CUSTOMER_TBL
      WHERE CUST_ID =
              (SELECT CUST_ID
               FROM ORDERS_TBL
               WHERE ORD_NUM = '16C17');
```

**b.**

```
SELECT EMP_ID, SALARY
      FROM EMPLOYEE_PAY_TBL
      WHERE SALARY BETWEEN '20000'
              AND (SELECT SALARY
                   FROM EMPLOYEE_ID
                   WHERE SALARY = '40000');
```

**c.**

```
UPDATE PRODUCTS_TBL
      SET COST = 1.15
      WHERE CUST_ID =
              (SELECT CUST_ID
               FROM ORDERS_TBL
               WHERE ORD_NUM = '32A132');
```



4. 下面语句执行的结果是什么？

```
DELETE FROM EMPLOYEE_TBL  
WHERE EMP_ID IN  
      (SELECT EMP_ID  
       FROM EMPLOYEE_PAY_TBL);
```

### 14.7.2 练习

1. 编写SQL的子查询代码，与书中提供的进行比较。使用下面的表来完成练习。

```

EMPLOYEE_TBL
EMP_ID      VARCHAR(9)      NOT NULL      primary key
LAST_NAME   VARCHAR(15)     NOT NULL
FIRST_NAME  VARCHAR(15)     NOT NULL
MIDDLE_NAME VARCHAR(15)
ADDRESS     VARCHAR(30)    NOT NULL
CITY        VARCHAR(15)   NOT NULL
STATE       VARCHAR(2)    NOT NULL
ZIP         INTEGER(5)     NOT NULL
PHONE       VARCHAR(10)
PAGER       VARCHAR(10)

EMPLOYEE_PAY_TBL
EMP_ID      VARCHAR(9)      NOT NULL      primary key
POSITION    VARCHAR(15)     NOT NULL
DATE_HIRE   DATETIME
PAY_RATE    DECIMAL(4,2)   NOT NULL
DATE_LAST_RAISE DATETIME
CONSTRAINT EMP_FK FOREIGN KEY (EMP_ID_ REFERENCES
EMPLOYEE_TBL (EMP_ID)

CUSTOMER_TBL
CUST_ID     VARCHAR(10)    NOT NULL      primary key
CUST_NAME   VARCHAR(30)    NOT NULL
CUST_ADDRESS VARCHAR(20)   NOT NULL
CUST_CITY   VARCHAR(15)    NOT NULL
CUST_STATE  VARCHAR(2)      NOT NULL
CUST_ZIP    INTEGER(5)     NOT NULL
CUST_PHONE  INTEGER(10)
CUST_FAX    INTEGER(10)

ORDERS_TBL
ORD_NUM     VARCHAR(10)    NOT NULL      primary key
CUST_ID     VARCHAR(10)    NOT NULL
PROD_ID     VARCHAR(10)    NOT NULL
QTY         INTEGER(6)     NOT NULL
ORD_DATE    DATETIME

PRODUCTS_TBL
PROD_ID     VARCHAR(10)    NOT NULL      primary key
PROD_DESC   VARCHAR(40)    NOT NULL
COST        DECIMAL(6,2)   NOT NULL

```

2. 使用子查询编写一个SQL语句来更新表CUSTOMER\_TBL，找到ORD\_NUM列中订单号码为 23E934的顾客，把顾客名称修改为

DAVIDS MARKET。

3. 使用子查询编写一个SQL语句，返回小时工资高于 JOHN DOE 的全部雇员的姓名；JOHN DOE的雇员标识号码是 343559876。

4. 使用子查询编写一个SQL语句，列出所有价格高于全部产品平均价格的产品。

## 第15章 组合多个查询

本章的重点包括：

简介用于组合查询的操作符

何时对查询进行组合

GROUP BY子句与组合命令

ORDER BY与组合命令

如何获取准确的数据

本章介绍如何使用操作符UNION、UNION ALL、INTERSECT和EXCEPT把多个SQL查询组合为一个。同样的，这些操作符的实际使用方法请参考具体实现的文档。

### 15.1 单查询与组合查询

单查询是一个SELECT语句，而组合查询具有两个或多个SELECT语句。

组合查询由负责结合两个查询的操作符组成，下面的范例使用操作符UNION结合两个查询。

单个SQL语句的范例：

```
SELECT EMP_ID, SALARY, PAY_RATE
FROM EMPLOYEE_PAY_TBL
WHERE SALARY IS NOT NULL OR
PAY_RATE IS NOT NULL;
```

下面是同一个语句使用操作符UNION:

```
SELECT EMP_ID, SALARY
FROM EMPLOYEE_PAY_TBL
WHERE SALARY IS NOT NULL
UNION
SELECT EMP_ID, PAY_RATE
FROM EMPLOYEE_PAY_TBL
WHERE PAY_RATE IS NOT NULL;
```

上面的语句返回所有雇员的工资信息，包含月薪和小时工资。

组合操作符用于组合和限制两个SELECT语句的结果，它们可以返回或清除重复的记录。组合操作符可以获取不同字段里的类似数据。

注意：**UNION**操作符如何起作用

第二个查询的输出结果里有两个列标题：**EMP\_ID** 和 **SALARY**，每个人的工资都列在**SALARY**之下。在使用UNION操作符时，列标题是由SELECT语句里的字段名称或字段别名决定的。

组合查询可以把多个查询的结果组合为一个数据集，而且通常比使用复杂条件的单查询更容易编写。另外，组合查询对于数据检索也具有更强的灵活性。

## [15.2 组合查询操作符](#)

不同数据库厂商提供的组合操作符略有不同。ANSI标准包括UNION、UNION ALL、EXCEPT和INTERSECT，下面的小节将分别讨论这些操作符。

### 15.2.1 UNION

UNION 操作符可以组合两个或多个 SELECT 语句的结果，不包含重复的记录。换句话说，如果某行的输出存在于一个查询结果里，那么其他查询结果同一行的记录就不会再输出了。在使用UNION操作符时，每个SELECT语句里必须选择同样数量的字段、同样数量的字段表达式、同样的数据类型、同样的次序——但长度不必一样。

语法如下：

```
SELECT COLUMN1 [, COLUMN2 ]  
FROM TABLE1 [, TABLE2 ]  
[ WHERE ]  
UNION  
SELECT COLUMN1 [, COLUMN2 ]  
FROM TABLE1 [, TABLE2 ]  
[ WHERE ]
```

比如下面这个范例：

```
SELECT EMP_ID FROM EMPLOYEE_TBL  
UNION  
SELECT EMP_ID FROM EMPLOYEE_PAY_TBL;
```

雇员ID在两个表里都存在，但在结果里只出现一次。

本章的范例由从两个表获取数据的简单SELECT语句开始：

```
SELECT PROD_DESC FROM PRODUCTS_TBL;
```

```
PROD_DESC
```

```
-----  
WITCH COSTUME  
PLASTIC PUMPKIN 18 INCH  
FALSE PARAFFIN TEETH  
LIGHTED LANTERNS  
ASSORTED COSTUMES  
CANDY CORN  
PUMPKIN CANDY  
PLASTIC SPIDERS  
ASSORTED MASKS  
KEY CHAIN  
OAK BOOKSHELF
```

```
11 rows selected.
```

```
SELECT PROD_DESC FROM PRODUCTS_TMP;
```

```
PROD_DESC
```

```
-----  
WITCH COSTUME  
PLASTIC PUMPKIN 18 INCH  
FALSE PARAFFIN TEETH  
LIGHTED LANTERNS  
ASSORTED COSTUMES  
CANDY CORN  
PUMPKIN CANDY  
PLASTIC SPIDERS  
ASSORTED MASKS  
KEY CHAIN  
OAK BOOKSHELF
```

```
11 rows selected.
```

现在利用UNION操作符组合上述两个查询，构造一个组合查询：

```
SELECT PROD_DESC FROM PRODUCTS_TBL
UNION
SELECT PROD_DESC FROM PRODUCTS_TMP;
```

```
PROD_DESC
-----
ASSORTED COSTUMES
ASSORTED MASKS
CANDY CORN
FALSE PARAFFIN TEETH
LIGHTED LANTERNS
PLASTIC PUMPKIN 18 INCH
PLASTIC SPIDERS
PUMPKIN CANDY
WITCH COSTUME
KEY CHAIN
OAK BOOKSHELF

11 rows selected.
```

注意：创建表**PRODUCTS\_TBL**

表PRODUCTS\_TBL是在第3章里创建的。

第一个查询返回11条数据，第二个查询返回11条数据，但使用UNION操作符组合两个查询之后只返回了11条数据，这是因为UNION不会返回重复的数据。

下面的范例使用UNION操作符组合两个不相关的查询：

```
SELECT PROD_DESC FROM PRODUCTS_TBL
UNION
SELECT LAST_NAME FROM EMPLOYEE_TBL;
```

```
PROD_DESC
-----
ASSORTED COSTUMES
ASSORTED MASKS
CANDY CORN
FALSE PARAFFIN TEETH
GLASS
KEY CHAIN
LIGHTED LANTERNS
OAK BOOKSHELF
PLASTIC PUMPKIN 18 INCH
PLASTIC SPIDERS
PLEW
PUMPKIN CANDY
SPURGEON
STEPHENS
WALLACE
WITCH COSTUME

16 rows selected.
```

PROD\_DESC和LAST\_NAME的值被列在一起，列标题来自于第一个查询的字段名称。

### **15.2.2 UNION ALL**

UNION ALL操作符可以组合两个SELECT语句的结果，并且包含重复的结果。其使用规则与UNION一样，它与UNION基本上是一样的，只是一个返回重复的结果，一个不返回。

基本语法如下所示：



```
SELECT COLUMN1 [, COLUMN2 ]  
FROM TABLE1 [, TABLE2 ]  
[ WHERE ]  
UNION ALL  
SELECT COLUMN1 [, COLUMN2 ]  
FROM TABLE1 [, TABLE2 ]  
[ WHERE ]
```

下面这个SQL语句返回全部雇员的ID，并且包含重复的记录：

```
SELECT EMP_ID FROM EMPLOYEE_TBL  
UNION ALL  
SELECT EMP_ID FROM EMPLOYEE_PAY_TBL
```

下面是使用UNION ALL操作符改写前一小节的组合查询：

```
SELECT PROD_DESC FROM PRODUCTS_TBL
UNION ALL
SELECT PROD_DESC FROM PRODUCTS_TMP;
```

```
PROD_DESC
-----
WITCH COSTUME
PLASTIC PUMPKIN 18 INCH
FALSE PARAFFIN TEETH
LIGHTED LANTERNS
ASSORTED COSTUMES
CANDY CORN
PUMPKIN CANDY
PLASTIC SPIDERS
ASSORTED MASKS

KEY CHAIN
OAK BOOKSHELF
WITCH COSTUME
PLASTIC PUMPKIN 18 INCH
FALSE PARAFFIN TEETH
LIGHTED LANTERNS
ASSORTED COSTUMES
CANDY CORN
PUMPKIN CANDY
PLASTIC SPIDERS
ASSORTED MASKS
KEY CHAIN
OAK BOOKSHELF
```

```
22 rows selected.
```

因为UNION ALL操作符会返回重复的数据，所以这个查询返回了22条记录（11+11）。

### [15.2.3 INTERSECT](#)

INTERSECT 可以组合两个 SELECT 语句，但只返回第一个 SELECT 语句里与第二个SELECT语句里一样的记录。其使用规则与

UNION操作符一样。目前MySQL5.0尚不支持INTERSECT，但SQL Server和Oracle全都提供支持。

基本语法如下所示：

```
SELECT COLUMN1 [, COLUMN2 ]  
FROM TABLE1 [, TABLE2 ]  
[ WHERE ]  
INTERSECT  
SELECT COLUMN1 [, COLUMN2 ]  
FROM TABLE1 [, TABLE2 ]  
[ WHERE ]
```

范例如下：

```
SELECT CUST_ID FROM CUSTOMER_TBL  
INTERSECT  
SELECT CUST_ID FROM ORDERS_TBL;
```

前面这个SQL语句返回具有订单的顾客的ID。

下面的范例使用INTERSECT组合两个查询：

```
SELECT PROD_DESC FROM PRODUCTS_TBL  
INTERSECT  
SELECT PROD_DESC FROM PRODUCTS_TMP;
```

```
PROD_DESC  
-----  
ASSORTED COSTUMES  
ASSORTED MASKS  
CANDY CORN  
FALSE PARAFFIN TEETH  
KEY CHAIN  
LIGHTED LANTERNS  
OAK BOOKSHELF  
PLASTIC PUMPKIN 18 INCH  
PLASTIC SPIDERS  
PUMPKIN CANDY  
WITCH COSTUME  
  
11 rows selected.
```

这里只返回了11条记录，因为两个查询之间只有11条记录是一样的。

#### **15.2.4 EXCEPT**

EXCEPT 操作符组合两个 SELECT 语句，返回第一个 SELECT 语句里有但第二个SELECT语句里没有的记录。同样的，它的使用规则与 UNION操作符一样。目前MySQL并不支持EXCEPT。而在Oracle中，则使用MINUS操作符来实现同样的功能。

其语法如下所示：

```

SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
EXCEPT
SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]

```

观察下面SQL Server实现中的范例：

```

SELECT PROD_DESC FROM PRODUCTS_TBL
EXCEPT
SELECT PROD_DESC FROM PRODUCTS_TMP;

PROD_DESC
-----
PLASTIC PUMPKIN 18 INCH
PLASTIC SPIDERS
PUMPKIN CANDY

3 rows selected.

```

根据结果可以了解到，有3条记录存在于第一个查询的结果且不存在于第二个查询的结果。

下面的范例展示了以MINUS代替EXCEPT。

```

SELECT PROD_DESC FROM PRODUCTS_TBL
MINUS
SELECT PROD_DESC FROM PRODUCTS_TMP;

PROD_DESC
-----
PLASTIC PUMPKIN 18 INCH
PLASTIC SPIDERS
PUMPKIN CANDY

3 rows selected.

```

### 15.3 组合查询里使用ORDER BY

ORDER BY子句可以用于组合查询，但它只能用于对全部查询结果的排序，因此组合查询里虽然可能包含多个查询或SELECT语句，但只能有一个ORDER BY子句，而且它只能以别名或数字来引用字段。

其语法如下所示：

```
SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
OPERATOR{UNION | EXCEPT | INTERSECT | UNION ALL}
SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
[ ORDER BY ]
```

下面这个范例从EMPLOYEE\_TBL表和EMPLOYEE\_PAY\_TBL表中返回雇员ID，但是不显示重复记录，返回结果根据EMP\_ID排序：

```
EMP_ID:
SELECT EMP_ID FROM EMPLOYEE_TBL
UNION
SELECT EMP_ID FROM EMPLOYEE_PAY_TBL
ORDER BY 1;
```

注意：在**ORDER BY**子句中使用数字

ORDER BY子句里的字段是以数字1进行引用的，没有什么实际的字段名称。

组合查询的结果以每个查询的第一个字段进行排序。在排序之后，重复的记录就很明显了。

下面的范例在组合查询里使用ORDER BY子句。如果排序的字段在全部查询语句里都具有相同的名称，它的名称就可以用于ORDER BY子

句里。

```
SELECT PROD_DESC FROM PRODUCTS_TBL  
UNION  
SELECT PROD_DESC FROM PRODUCTS_TBL  
ORDER BY PROD_DESC;
```

```
PROD_DESC  
-----  
ASSORTED COSTUMES  
ASSORTED MASKS  
CANDY CORN  
FALSE PARAFFIN TEETH  
KEY CHAIN  
LIGHTED LANTERNS  
OAK BOOKSHELF  
PLASTIC PUMPKIN 18 INCH  
PLASTIC SPIDERS  
PUMPKIN CANDY  
WITCH COSTUME  
  
11 rows selected.
```

下面的查询在ORDER BY子句里以数据代表字段：

```
SELECT PROD_DESC FROM PRODUCTS_TBL  
UNION  
SELECT PROD_DESC FROM PRODUCTS_TBL;
```

```
PROD_DESC
```

```
-----
```

```
ASSORTED COSTUMES  
ASSORTED MASKS  
CANDY CORN  
FALSE PARAFFIN TEETH  
KEY CHAIN  
LIGHTED LANTERNS  
OAK BOOKSHELF  
PLASTIC PUMPKIN 18 INCH  
PLASTIC SPIDERS  
PUMPKIN CANDY  
WITCH COSTUME
```

```
11 rows selected.
```

## [15.4 组合查询里使用GROUP BY](#)

与ORDER BY不同的是，GROUP BY子句可以用于组合查询中的每一个 SELECT语句，也可以用于全部查询结果。另外，HAVING子句也可以用于组合查询里的每个SELECT语句。

其语法如下所示：



```

SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
[ GROUP BY ]
[ HAVING ]
OPERATOR {UNION | EXCEPT | INTERSECT | UNION ALL}
SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
[ GROUP BY ]
[ HAVING ]
[ ORDER BY ]

```

下面的查询利用一个字符串代表顾客记录、雇员记录和产品记录。每个单独的查询就是统计表里的记录总数。GROUP BY子句用于把整个结果根据第一个字段进行分组。

```

SELECT 'CUSTOMERS' TYPE, COUNT(*)
FROM CUSTOMER_TBL
UNION
SELECT 'EMPLOYEES' TYPE, COUNT(*)
FROM EMPLOYEE_TBL
UNION
SELECT 'PRODUCTS' TYPE, COUNT(*)
FROM PRODUCTS_TBL
GROUP BY 1;

```

TYPE	COUNT(*)
CUSTOMERS	15
EMPLOYEES	6
PRODUCTS	9

3 rows selected.

下面的查询与前一个一样，只是使用了ORDER BY子句：

```

SELECT 'CUSTOMERS' TYPE, COUNT(*)
FROM CUSTOMER_TBL
UNION
SELECT 'EMPLOYEES' TYPE, COUNT(*)
FROM EMPLOYEE_TBL
UNION
SELECT 'PRODUCTS' TYPE, COUNT(*)
FROM PRODUCTS_TBL
GROUP BY 1
ORDER BY 2;

```

TYPE	COUNT(*)
EMPLOYEES	6
PRODUCTS	9
CUSTOMERS	15

3 rows selected.

它根据每个表里的第二列进行排序，因此输出结果根据总数从小到大排列。

注意：错误数据

不完整的查询返回结果被称为错误数据。

## 15.5 获取准确的数据

使用组合查询时要小心。在使用INTERSECT操作符时，如果第一个查询的SELECT语句有问题，就可能会得到不正确或不完整的数据。另外，在使用UNION和UNION ALL操作符时，要考虑是否需要返回重复的数据。那EXCEPT呢？我们是否需要不存在于第二个查询里的数据？很明显，组合查询里的错误组合操作符或单个查询的次序有误都会导致返回不正确的数据。

## 15.6 小结

本章介绍了组合查询。之前介绍的SQL语句都是构成单个查询，而组合查询可以让多个查询一起返回一个统一的数据集。这里讨论的组合操作符包括 UNION、UNION ALL、INTERSECT和 EXCEPT (MINUS)。UNION返回两个查询的结果，不包含重复记录。UNION ALL会返回两个查询的全部结果，不管数据是否重复。INTERSECT返回两个查询结果中一样的记录。EXCEPT (MINUS) 返回一个查询结果中不存在于另一个查询结果的记录。组合查询具有很大的灵活性，能够满足各种查询的要求。如果不使用组合查询，可能需要很复杂的查询语句才能达到同样的结果。

## [15.7 问与答](#)

问：组合查询中的**GROUP BY**子句如何引用字段？

答：如果被引用的字段在所有查询里都是相同的名称，就可以直接使用字段名称进行引用；否则可以使用字段在SELECT语句里的次序号码进行引用。

问：在使用**EXCEPT**操作符时，如果颠倒**SELECT**语句的次序是否会改变输出结果呢？

答：是的。在使用EXCEPT或MINUS操作符时，单个查询的次序是很重要的。返回的数据是存在于第一个查询结果且不存在于第二个查询结果的记录，所以改变单个查询的次序肯定会改变结果。

问：组合查询里的单个查询的字段是否一定要具有同样的数据类型和长度？

答：不，只有数据类型要求是一样的，长度可以不同。

问：使用**UNION**操作符时，字段名称是由什么决定的？

答：在使用UNION操作符时，第一个查询决定了输出的字段名称。

## 15.8 实践

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### 15.8.1 测验

在下面的练习里使用INTERSECT或EXCEPT操作符时，请参考本章介绍的语法。请注意，MySQL目前还不支持这两个操作符。

1. 下面组合查询的语法正确吗？如果不正确，请修改它们。它们使用的表EMPLOYEE\_TBL和EMPLOYEE\_PAY\_TBL如下所示：

```
EMPLOYEE_TBL
EMP_ID      VARCHAR(9)      NOT NULL,
LAST_NAME   VARCHAR(15)    NOT NULL,
FIRST_NAME  VARCHAR(15)    NOT NULL,
MIDDLE_NAME VARCHAR(15),
ADDRESS     VARCHAR(30)    NOT NULL,
CITY        VARCHAR(15)    NOT NULL,
STATE       VARCHAR(2)     NOT NULL,
ZIP         INTEGER(5)     NOT NULL,
PHONE       VARCHAR(10),
PAGER       VARCHAR(10),
CONSTRAINT EMP_PK PRIMARY KEY (EMP_ID)
```

```
EMPLOYEE_PAY_TBL
EMP_ID      VARCHAR(9)      NOT NULL    primary key,
POSITION    VARCHAR(15)    NOT NULL,
DATE_HIRE   DATETIME,
PAY_RATE    DECIMAL(4,2)   NOT NULL,
DATE_LAST_RAISE DATE,
SALARY      DECIMAL(8,2),
BONUS       DECIMAL(6,2),
CONSTRAINT EMP_FK FOREIGN KEY (EMP_ID)
REFERENCES EMPLOYEE_TBL (EMP_ID)
```

**a.**

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME  
FROM EMPLOYEE_TBL  
UNION  
SELECT EMP_ID, POSITION, DATE_HIRE  
FROM EMPLOYEE_PAY_TBL;
```

**b.**

```
SELECT EMP_ID FROM EMPLOYEE_TBL  
UNION ALL  
SELECT EMP_ID FROM EMPLOYEE_PAY_TBL  
ORDER BY EMP_ID;
```

**c.**

```
SELECT EMP_ID FROM EMPLOYEE_PAY_TBL  
INTERSECT  
SELECT EMP_ID FROM EMPLOYEE_TBL  
ORDER BY 1;
```

2. 匹配操作符与相应的描述。

描述 操作符

- a. 显示重复记录 UNION
- b. 返回第一个查询里与第二个查询匹配的结果 INTERSECT
- c. 返回不重复的记录 UNION ALL
- d. 返回第一个查询里有但第二个查询没有的结果 EXCEPT

### 15.8.2 练习

下面的练习请参考本章介绍的语法。由于 MySQL 不支持本章介绍的两个操作符，所以请自行编写查询语句，并与书中提供的进行比较。

使用的表CUSTOMER\_TBL和ORDERS\_TBL如下所示：

```

CUSTOMER_TBL
CUST_IN      VARCHAR(10)    NOT NULL      primary key,
CUST_NAME    VARCHAR(30)    NOT NULL,
CUST_ADDRESS VARCHAR(20)    NOT NULL,
CUST_CITY    VARCHAR(15)    NOT NULL,
CUST_STATE   VARCHAR(2)     NOT NULL,
CUST_ZIP     INTEGER(5)     NOT NULL,
CUST_PHONE   INTEGER(10),
CUST_FAX     INTEGER(10)

ORDERS_TBL
ORD_NUM      VARCHAR(10)    NOT NULL      primary key,
CUST_ID      VARCHAR(10)    NOT NULL,
PROD_ID      VARCHAR(10)    NOT NULL,
QTY          INTEGER(6)     NOT NULL,
ORD_DATE     DATETIME

```

1. 编写一个组合查询，返回下了订单的顾客。
2. 编写一个组合查询，返回没有下订单的顾客。

## [第五部分 SQL性能调整](#)

第16章 利用索引改善性能

第17章 改善数据库性能

### [第16章 利用索引改善性能](#)

本章的重点包括：

索引如何工作

如何创建索引

不同类型的索引

何时使用索引

何时不使用索引

本章介绍如何通过创建和使用索引来改善 SQL语句的性能，首先介绍 CREATE INDEX命令，然后介绍如何使用表里的索引。

#### [16.1 什么是索引](#)

简单来说，索引就是一个指针，指向表里的数据。数据库里的索引与图书中的索引十分类似。举例来说，如果想查阅书中关于某个主题的内容，我们首先会查看索引，其中会以字母顺序列出全部主题，告诉我们一个或多个特定的书页号码。索引在数据库里也起到这样的作用，指向数据在表里的准确物理位置。实际上，我们被引导到数据在数据库底层文件里的位置，但从表面上来看，我们是在引用一个表。

在查找信息时，逐页寻找快呢，还是查看索引来了解准确页码快呢？当然，使用索引是最有效的方法。当书很厚时，这样做会节省大量时间。假设书只有几页，那么直接查找信息可能会比先看索引再返回到

某页更快一些。当数据库没有索引时，它所进行的操作通常被称为全表扫描，就像是逐页翻看一本书。关于全表扫描的具体介绍请见第17章。

索引通常与相应的表是分开保存的，其主要目的是提高数据检索的性能。索引的创建与删除不会影响到数据本身，但会影响数据检索的速度。索引也会占据物理存储空间，而且可能会比表本身还大。因此在考虑数据库的存储空间时，需要考虑索引要占用的空间。

## 16.2 索引是如何工作的

索引在创建之后，用于记录与被索引字段相关联的位置值。当表里添加新数据时，索引里也会添加新项。当数据库执行查询，而且WHERE条件里指定的字段已经设置了索引时，数据库会首先在索引里搜索WHERE子句里指定的值。如果在索引里找到了这个值，索引就可以返回被搜索数据在表里的实际位置。图16.1展示了索引的工作过程。

假设执行了如下查询：

```
SELECT *  
FROM TABLE_NAME  
WHERE NAME = 'SMITH';
```

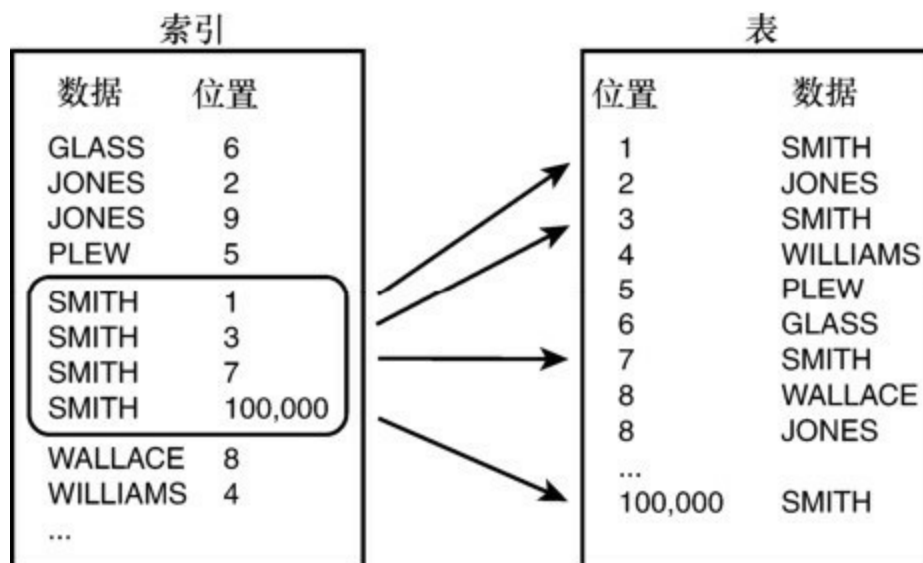




图16.1 使用索引访问表

如图16.1所示，这里引用了索引NAME来寻找‘SMITH’的位置；在找到了位置之后，数据就能迅速地从表里检索出来。在索引里，数据（本例中是姓名）是按字母顺序排序的。

注意：索引的不同创建方式

在某些实现里，可以在创建表的过程中创建索引。但大多数实现提供了一个单独的命令来创建索引，其详细语法请参考具体的文档。

如果表里没有索引，在执行同样这个查询时，数据库就会进行全表扫描，也就是说表里的每行数据都会被读取来获取NAME字段等于‘SMITH’的记录。

索引通常以一种树形结构保存信息，因此速度比较快。假设我们对一个书名列表设置了索引，这个索引具有一个根节点，也就是每个查询的起始点。根节点具有分支，在本例中可以有两个分支，一个代表字母A到L，另一个代表字母M到Z。如果要查询以字母M开头的书名，我们就会从根节点进入索引，并且立即转到包含字母M到Z的分支。这种方式可以消除大约一半的可能性，从而用更短的时间找到准确的书名。

### 16.3 CREATE INDEX命令

像SQL里的其他语句一样，创建索引的语句在不同关系型数据库实现里也是不同的，大多数实现使用CREATE INDEX语句：

```
CREATE INDEX INDEX_NAME ON TABLE_NAME
```

不同厂商的CREATE INDEX语句在选项方面有不少差别，有些实现允许指定存储子句（像CREATE TABLE语句）、允许排序（DESC||ASC）、允许使用簇。详细语法请查看具体实现的文档。

## [16.4 索引的类型](#)

数据库里的表可以创建多种类型的索引，它们的目标是一样的：通过提高数据检索速度来改善数据库性能。本章介绍单字段索引、组合索引和唯一索引。

### [16.4.1 单字段索引](#)

提示：最有效的单字段索引

如果某个字段经常在WHERE子句作为单独的查询条件，它的单字段索引是最有效的。适合作为单字段索引的值有个人标识号码、序列号或系统指派的键值。

对单个字段的索引是索引中最简单、最常见的形式。显然，单字段索引是基于一个字段创建的，其基本语法如下所示：

```
CREATE INDEX INDEX_NAME  
ON TABLE_NAME (COLUMN_NAME)
```

举例来说，如果想对表EMPLOYEE\_TBL里雇员的姓创建索引，相应的命令如下所示：

```
CREATE INDEX NAME_IDX  
ON EMPLOYEE_TBL (LAST_NAME);
```

### [16.4.2 唯一索引](#)

唯一索引用于改善性能和保证数据完整性。唯一索引不允许表里具有重复值，除此之外，它与普通索引的功能一样。其语法如下所示：

```
CREATE UNIQUE INDEX INDEX_NAME  
ON TABLE_NAME (COLUMN_NAME)
```

如果想对表EMPLOYEE\_TBL里雇员的姓创建唯一索引，相应的命令如下所示：

```
CREATE UNIQUE INDEX NAME_IDX  
ON EMPLOYEE_TBL (LAST_NAME);
```

这个索引唯一需要注意的问题是，表EMPLOYEE\_TBL里每个人的姓都必须是唯一的，这通常是不现实的。但是，像个人社会保险号码这样的字段可以设置为唯一索引，因为每个人的这个号码都是唯一的。

有人也许会问，如果雇员的社会保险号码是表的主键，那应该怎么办呢？当我们定义表的主键时，一个默认的索引就会被创建。但是，公司会使用自己编制的号码作为雇员ID，同时使用雇员的SSN用于纳税。通常我们会对这个字段设置索引，确保它在每条记录里都具有唯一的值。

对于类似索引这种对象，一个比较可取的方法是，在创建数据库结构的同时，基于空白表来创建索引。这样做可以确保后续输入的数据完全满足用户的要求。如果要在既有数据中创建索引，就必须进行相应的分析工作，来确定是否需要调整数据以便符合索引的要求。

### 16.4.3 组合索引

组合索引是基于一个表里两个或多个字段的索引。在创建组合索引时，我们要考虑性能的问题，因为字段在索引里的次序对数据检索速度有很大的影响。一般来说，最具有限制的值应该排在前面，从而得到最好的性能。但是，总是会在查询里指定的字段应该放在首位。组合索引的语法如下所示：

```
CREATE INDEX INDEX_NAME  
ON TABLE_NAME (COLUMN1, COLUMN2)
```

组合索引的范例如下所示：

```
CREATE INDEX ORD_IDX  
ON ORDERS_TBL (CUST_ID, PROD_ID);
```

在这个范例里，我们基于表ORDERS\_TBL里的两个字段（CUST\_ID和PROD\_ID）创建组合索引。这是因为我们认为这两个字段经常会在查询的WHERE子句里联合使用。

注意：唯一索引的相关规则

唯一索引只能用于在表里没有重复值的字段。换句话说，如果现有表已经包含被索引关键字的记录，就不能再对它创建唯一索引了。此外，允许NULL值的字段上也不能创建唯一索引。如果不满足上述规则，那么创建语句就无法运行成功。

在选择是使用单字段索引还是组合索引时，要考虑在查询的WHERE子句里最经常使用什么字段。如果经常只使用一个字段，单字段索引就是最适合的；如果经常使用两个或多个字段，组合索引就是最好的索引。

#### [16.4.4 隐含索引](#)

隐含索引是数据库服务程序在创建对象时自动创建的。比如，数据库会为主键约束和唯一性约束自动创建索引。

为什么给这些约束自动创建索引？从一个数据库服务程序的角度来看，当用户向数据库添加一个新产品时，产品标识是表里的主键，表示它必须是唯一值。为了有效地检查新值在数以百计甚至是数以千计的记录里是唯一的，表里的产品标识必须被索引。因此，在创建主键或唯一性约束时，数据库会自动为它们创建索引。

提示：最有效的组合索引

对于经常在查询的WHERE子句里共同使用的字段，组合索引是最

有效的。

## [16.5 何时考虑使用索引](#)

唯一索引隐含地与主键共同实现主键的功能。外键经常用于与父表的结合，所以也适合设置索引。一般来说，大多数用于表结合的字段都应该设置索引。

经常在ORDER BY和GROUP BY里引用的字段也应该考虑设置索引。举例来说，如果根据个人姓名进行排序，对姓名字段设置索引会大有好处。它会对每个姓名自动按字母顺序排序，简化了实际的排序操作，提高了输出结果的速度。

另外，具有大量唯一值的字段，或是在WHERE子句里会返回很小部分记录的字段，都可以考虑设置索引。这主要是为了测试或避免错误。就像代码和数据库结构在投入使用之前需要反复进行测试一样，索引也是如此。我们应该用一些时间来尝试不同的索引组合、没有索引、单字段索引和组合索引。索引的使用没有什么固定的规则，需要对表的关系、查询和事务需求、数据本身有透彻的了解才能最有效地使用索引。

## [16.6 何时应该避免使用索引](#)

注意：要有事先规划

表和索引都应该进行事先的规划。不要认为使用索引就能解决所有的性能问题，索引可能根本不会改善性能（甚至可能降低性能）而只是占据磁盘空间。

虽然使用索引的初衷是提高数据库性能，但有时也要避免使用它们。下面是使用索引的方针。

索引不应该用于小规模表。因为查询索引会增加额外的查询时

间。对于小规模表，让搜索发动机进行全表搜索，往往比先查询索引的速度更快。

当字段用于WHERE子句作为过滤器会返回表里的大部分记录时，该字段就不适合设置索引。举例来说，图书里的索引不会包括像the或and这样的单词。

经常会被批量更新的表可以具有索引，但批量操作的性能会由于索引而降低。对于经常会被加载或批量操作的表来说，可以在执行批量操作之前去除索引，在完成操作之后再重新创建索引。这是因为当表里插入数据时，索引也会被更新，从而增加了额外的开销。

不应该对包含大量NULL值的字段设置索引。索引对在不同记录中包含不同数据的字段特别有效。字段中过多的NULL值会严重影响索引的运行效率。

经常被操作的字段不应该设置索引，因为对索引的维护会变得很繁重。

从图16.2可以看出，像性别这样的字段设置索引就没有什么好处。举例来说，向数据库提交如下查询：

```
SELECT *  
FROM TABLE_NAME  
WHERE GENDER = 'FEMALE';
```

从图16.2可以看出，在运行上述这个查询时，表与索引之间有一个持续的行为。由于WHERE GENDER = 'FEMALE'（或'MALE'）子句会返回大量记录，数据库服务程序必须持续地读取索引、然后读取表的内容、再读取索引、再读取表，如此反复。在这个范例里，由于表里的大部分数据肯定是要被读取的，所以使用全表扫描可能会效率更高。

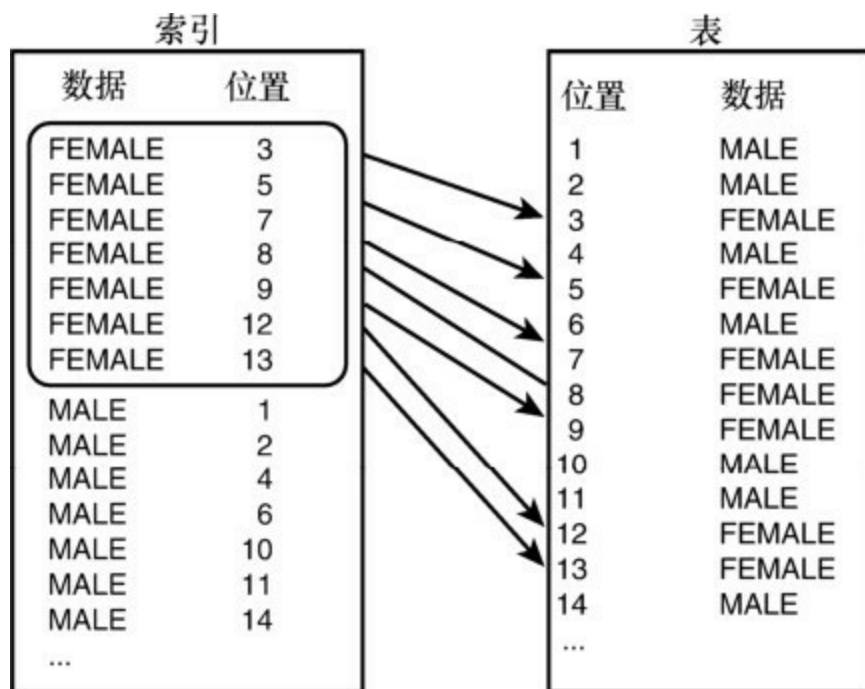


图16.2 低效索引的例子

警告：索引也会带来运行问题

对于特别长的关键字创建索引时要十分谨慎，因为大量I/O开销会不可避免地降低数据库性能。

一般来说，当字段作为查询里的条件会返回表里的大部分数据时，我们不会对它设置索引。换句话说，不要对像性别这样只包含很少不同值的字段设置索引。这通常被称为字段的基数，或数据的唯一性。高基数意味着很高的唯一性，比如像身份号码这样的数据。低基数的唯一性不高，比如像性别这样的字段。

## [16.7 修改索引](#)

创建索引后，也可以对其进行修改。其语法结构与CREATE INDEX类似。能够修改的内容在不同的数据库实现中有所不同，但基本上修改的都是字段、顺序等内容。其语法如下所示：



`ALTER INDEX INDEX_NAME`

对生产系统进行修改时需要特别小心。大部分情况下，对索引进行的修改操作会被马上执行，引起系统资源的额外消耗。此外，大部分数据库实现在进行索引修改的时候无法进行查询操作，从而会对系统的运行产生影响。

## [16.8 删除索引](#)

删除索引的方法相当简单，具体语法请参考相应的文档，但大多数实现使用**DROP**命令。在删除索引时要谨慎，因为性能可能会严重降低（或提高！）。其语法如下所示：

`DROP INDEX INDEX_NAME`

MySQL中的语法结构稍有不同，需要同时指定创建索引的表格：

`DROP INDEX INDEX_NAME ON TABLE_NAME`

删除索引的最常见原因是尝试改善性能。记住，在删除索引之后，我们还可以重新创建它。有时重建索引是为了减少碎片。在探索如何让数据库具有最佳性能时，调整索引是个必要的过程，其中可能包括创建索引、删除它、最后再重新创建它（经过修改或不修改）。

提示：小心使用索引

索引对于提高性能大有帮助，但在有些情况下也会降低性能。我们应该避免对只包含很少不同值的字段创建索引，比如性别、州名等。

注意：删除索引的语法差异

MySQL使用**ALTER TABLE**命令删除索引。也可以使用**DROP INDEX**命令，MySQL会将其映射为适当的**ALTER TABLE**命令。再次提醒，不同的SQL实现在语法方面可能会有所不同，特别是在处理索引



和数据存储的时候。

## [16.9 小结](#)

索引可以用于改善查询和事务的整体性能。数据库索引（有点像图书里的索引）可以迅速地从表里引用特定的数据。创建索引的最常用方法是使用CREATE INDEX命令。在不同的实现里有多种不同类型的索引，包括单字段索引、唯一索引和组合索引。在判断使用什么类型的索引时需要考虑多方面的因素，才能让它最好地满足数据库的需要。有效地使用索引通常需要有一定的经验、全面了解表的关系和数据，以及一点实践，设置索引时的一点点耐心可能会为以后的工作节约几分钟、几小时，甚至几天的时间。

## [16.10 问与答](#)

问：索引是否像表一样占据实际的空间？

答：是的。索引在数据库里占据物理空间。实际上，索引可能比所在的表更大。

问：如果为了让批处理工作更快地完成而删除了索引，需要多长时间才能重新创建索引？

答：这取决于多个因素，比如索引的大小、CPU利用率和计算机的性能。

问：全部索引都必须是唯一索引吗？

答：不是。唯一索引不允许存在重复值，而在表里有时是需要有重复值的。

## [16.11 实践](#)

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在

于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### 16.11.1 测验

1. 使用索引的主要缺点是什么？
2. 组合索引里的字段顺序为什么很重要？
3. 具有大量NULL值的字段是否应该设置索引？
4. 索引的主要作用是去除表里的重复数据吗？
5. 判断正误：使用组合索引主要是为了在索引里使用汇聚函数。
6. 基数是什么含义？什么样的字段可以被看作是高基数的？

### 16.11.2 练习

1. 判断在下列情况下是否应该使用索引，如果是，请选择索引的类型。
  - a. 字段很多，但表的规模相对较小。
  - b. 中等规模的表，不允许有重复值。
  - c. 多个字段，大规模的表，多个字段用在WHERE子句作为过滤器。
  - d. 大规模表，很多字段，大量数据操作。
2. 编写 SQL 语句，为表 EMPLOYEE\_PAY\_TBL 的 POSITION 字段创建名为 EP\_POSITION的索引。
3. 修改练习2所创建的索引，将其变成唯一索引。要为SALARY字段创建唯一索引，需要做些什么？编写并依次运行这些命令。
4. 研究本书里使用的表，根据用户可能对表进行的检索方式，判断哪些字段适合设置索引。
5. 在表 ORDERS\_TBL 上创建一个多字段索引，包含下列字段：

CUST\_ID、PROD\_ID和ORD\_DATE。

6. 在表里创建其他一些索引。

## [第17章 改善数据库性能](#)

本章的重点包括：

什么是SQL语句调整

数据库调整与SQL语句调整

格式化SQL语句

适当地结合表

最严格的条件

全表扫描

使用索引

避免使用OR和HAVING

避免大规模排序操作

本章介绍如何使用一些非常简单的方法调整SQL语句来获得最好的性能。

### [17.1 什么是SQL语句调整](#)

SQL语句调整是优化生成SQL语句的过程，从而以最有效和最高效的方式获得结果。首先是查询里元素的基本安排，因为简单的格式化过程就能够在语句优化中发挥很大作用。

SQL语句调整主要涉及调整语句的FROM和WHERE子句，因为数据库服务程序主要根据这两个子句执行查询。前面的课程已经介绍了FROM和WHERE子句的基础知识，现在就来介绍如何细致地调整它们来获得更好的结果，让用户更加满意。

## [17.2 数据库调整与SQL语句调整](#)

在继续介绍SQL语句调整之前，先要理解数据库调整与SQL语句调整之间的差别。

数据库调整是调整实际数据库的过程，包括分配内存、磁盘、CPU、I/O 和底层数据库进程，还涉及数据库结构本身的管理与操作，比如表和索引的设计与布局。另外，数据库调整通常会包括调整数据库体系来优化硬件的使用。实际上，在调整数据库时还要考虑其他很多因素，但这些任务通常是由数据库管理员（DBA）与系统管理员合作完成的。数据库调整的目标是确保数据库的设计能够最好地满足用户对数据库操作的需要。

SQL调整是调整访问数据库的SQL语句，这些语句包括数据库查询和事务操作，比如插入、更新和删除。SQL语句调整的目标是利用数据库和系统资源、索引，针对数据库的当前状态进行最有效的访问，从而减少对数据库执行查询所需的开销。

注意：两种调整缺一不可

为了在访问数据库时达到优化结果，数据库调整和SQL语句调整都需要进行。一个调整很差的数据库会极大地抵消SQL调整所付出的努力，反之亦然。在理想状态下，最好首先调整数据库，确保必要的字段都具有索引，然后再调整SQL代码。

## [17.3 格式化SQL语句](#)

格式化SQL语句听上去是个很显然的事情，但也值得一提。一个新手在构造SQL语句时很可能会忽略很多方面，下面的小节将进行讨论，它们有些是很明显的，有些则不是。

为提高可读性格式化SQL语句。

FROM子句里表的顺序。

最严格条件在WHERE子句里的位置。

结合条件在WHERE子句里的位置。

### 17.3.1 为提高可读性格式化SQL语句

注意：一切以最优化为目的

大多数关系型数据库实现里有一个名为“SQL 优化器”的东西，它可以执行SQL语句，并且基于SQL语句的构成方式和数据库里可用的索引来判断执行语句的最佳方式。这些优化器并不是都相同，具体情况请查看相应的文档，或是联系数据库管理员来了解优化器如何读取SQL代码。理解优化器的工作方式有助于有效地调整SQL语句。

为提高可读性格式化SQL语句是件很显然的事情，但很多SQL语句的书写方式并不那么整洁。虽然语句的整洁程度并不会影响实际的性能（数据库并不关心语句的外观是否整洁），但仔细地使用格式是调整语句的第一步。当我们以调整的眼光看待一个SQL语句时，让它具有很好的可读性总是首先要考虑的。如果语句很难看清，又如何能够判断它是否正确呢？

让语句具有良好可读性的基本规则如下所示。

每个子句都以新行开始。举例来说，让FROM子句位于与SELECT子句不同的行里，让WHERE子句位于与FROM子句不同的行里，以此类推。

当子句里的参数超过一行长度需要换行时，利用制表符（TAB）或空格来形成缩进。

以一致的方式使用制表符和空格。

当语句里使用多个表时，使用表的别名。在这种语句里使用表的全名来限定每个字段会让语句迅速变得冗长，让可读性降低。

如果SQL实现里允许使用注释，应该在语句里有节制地使用。注释是很好的文档，但过多的注释会让语句臃肿。

如果在SELECT语句里要使用多个字段，就让每个字段都从新行开始。

如果在FROM子句里要使用多个表，就让每个表名都从新行开始。

让WHERE子句里每个条件都以新行开始，这样就可以清晰地看到语句的所有条件及其次序。

下面是一个可读性很差的SQL语句：

```
SELECT CUSTOMER_TBL.CUST_ID, CUSTOMER_TBL.CUST_NAME,
CUSTOMER_TBL.CUST_PHONE, ORDERS_TBL.ORD_NUM, ORDERS_TBL.QTY
FROM CUSTOMER_TBL, ORDERS_TBL
WHERE CUSTOMER_TBL.CUST_ID = ORDERS_TBL.CUST_ID
AND ORDERS_TBL.QTY > 1 AND CUSTOMER_TBL.CUST_NAME LIKE 'G%'
ORDER BY CUSTOMER_TBL.CUST_NAME;
```

CUST_ID	CUST_NAME	CUST_PHONE	ORD_NUM	QTY
287	GAVINS PLACE	3172719991	18D778	10

1 row selected.

下面是格式化之后的语句，可读性明显提高：

```
SELECT C.CUST_ID,
       C.CUST_NAME,
       C.CUST_PHONE,
       O.ORD_NUM,
       O.QTY
FROM ORDERS_TBL O,
     CUSTOMER_TBL C
WHERE O.CUST_ID = C.CUST_ID
      AND O.QTY > 1
      AND C.CUST_NAME LIKE 'G%'
ORDER BY 2;
```

CUST_ID	CUST_NAME	CUST_PHONE	ORD_NUM	QTY
287	GAVINS PLACE	3172719991	18D778	10

1 row selected.

这两个语句完全一样，但第二个语句具有更好的可读性。通过使用

表的别名（在FROM子句里定义），第二个语句得到了极大的简化。同时使用空格对齐每个子句里的元素，让每个子句十分明显。

注意：在使用多个表的同时确保性能

当 FROM 子句里列出了多个表时，请查看具体实现的文档来了解有关提高性能的技巧。

再强调一次，虽然提高语句的可读性并不会直接改善它的性能，但这样会帮助我们更方便地修改和调整很长和很复杂的语句。现在我们可以轻松地看到被选择的字段、所使用的表、所执行的表结合和查询的条件。

### 17.3.2 FROM子句里的表

FROM子句里表的安排或次序对性能有很大影响，取决于优化器如何读取SQL语句。举例来说，把较小的表列在前面，把较大的表列在后面，就会获得更好的性能。有些经验丰富的用户发现把较大的表列在FROM子句的最后面可以得到更好的效率。

下面是FROM子句的一个范例：

```
FROM SMALLEST TABLE,  
      LARGEST TABLE
```

注意：创建编码标准

在多人编程环境里，创建编码标准是特别重要的。如果全部代码具有一致的格式，就可以更好地管理共享代码及修改代码。

### 17.3.3 结合条件的次序

第 13 章曾经介绍过，大多数结合使用一个基表链接到具有一个或多个共有字段的其他表。基表是主表，查询里的大多数或全部表都与它结合。在WHERE子句里，来自基表的字段一般放到结合操作的右侧，要被结合的表通常按照从小到大的次序排列，就像FROM子句里表的排



列顺序一样。

如果没有基表，那表就应该从小到大排列，让最大的表位于WHERE子句里结合操作的右侧。结合条件应该位于WHERE子句的最前面，其后才是过滤条件，如下所示：

FROM TABLE1,	Smallest table
TABLE2,	to
TABLE3	Largest table, also base table
WHERE TABLE1.COLUMN = TABLE3.COLUMN	Join condition
AND TABLE2.COLUMN = TABLE3.COLUMN	Join condition
[ AND CONDITION1 ]	Filter condition
[ AND CONDITION2 ]	Filter condition

提示：严格限制结合操作的条件

由于结合操作通常会从表里返回大部分数据，所以结合条件应该在更严格的条件之后再生效。

在这个范例里，TABLE3是基表，TABLE1和TABLE2结合到TABLE3。

#### [17.3.4 最严格条件](#)

最严格条件通常是 SQL 查询达到最优性能的关键因素。什么是最严格的条件？它是WHERE子句里返回最少记录的条件。与之相反，最宽松的条件就是语句里返回最多记录的条件。在这里我们重点关注最严格的条件，因为它对查询返回的数据进行了最大限度的过滤。

我们应该让SQL优化器首先计算最严格条件，因为它会返回最小的数据子集，从而减小查询的开销。最严格条件的位置取决于优化器的工作方式，有时优化器从WHERE子句的底部开始读取，因此需要把最严格条件放到WHERE子句的末尾，从而让优化器首先读取它。下面的例子展示了如何根据约束条件来构造 WHERE 子句，以及如何根据表的体积来构造FROM子句。



FROM TABLE1,	Smallest table
TABLE2,	to
TABLE3	Largest table, also base table
WHERE TABLE1.COLUMN = TABLE3.COLUMN	Join condition
AND TABLE2.COLUMN = TABLE3.COLUMN	Join condition
[ AND CONDITION1 ]	Least restrictive
[ AND CONDITION2 ]	Most restrictive

提示：对**WHERE**子句进行测试

如果不知道具体实现的SQL优化器如何工作、DBA也不知情、也没有足够的文档资料，我们可以执行一个需要一定时间的大型查询，然后重新排列WHERE子句里的条件，记录每次查询执行所需的时间。采取这种方法，不用几次测试就可以判断出优化器读取WHERE子句的方向。为了在测试中获得更准确的结果，最好在测试时关闭数据库缓存。

下面是一个虚构表的测试范例：

表	TEST
记录数量	95 867
条件	WHERE LAST_NAME = 'SMITH' 返回 2 000 条记录 WHERE CITY = 'INDIANAPOLIS' 返回 30 000 记录
最严格条件是	WHERE LAST_NAME = 'SMITH'

下面是第一个查询：

```

SELECT COUNT(*)
FROM TEST
WHERE LAST_NAME = 'SMITH'
AND CITY = 'INDIANAPOLIS';

COUNT(*)
-----
1,024

```

下面是第二个查询：

```

SELECT COUNT(*)
FROM TEST
WHERE CITY = 'INDIANAPOLIS'
      AND LAST_NAME = 'SMITH';

COUNT(*)
-----
      1,024

```

假设第一个查询用了20秒，第二个查询用10秒。由于第二个查询速度比较快，而且在它的WHERE子句里，最严格条件位于最后的位置，所以我们可以认为优化器从WHERE子句的底部开始读取条件。

注意：使用索引字段

从实践总结出来的经验表明，最好使用具有索引的字段作为查询里的最严格条件。索引通常会改善查询的性能。

## 17.4 全表扫描

在没有使用索引时，或是SQL语句所使用的表没有索引时，就会发生全表扫描。一般来说，全表扫描返回数据的速度要明显比使用索引慢。表越大，全表扫描返回数据的速度就越慢。查询优化器会决定在执行SQL语句时是否使用索引，而大多数情况会使用索引（如果存在）。

有些实现具有复杂的查询优化器，可以决定是否应该使用索引。这种判断基于从数据库对象上收集的统计信息，比如对象的规模、索引字段在指定条件下返回的记录数量等。关于优化器的这种判决能力请查看具体实现的文档。

在读取大规模的表时，应该避免进行全表扫描。举例来说，当读取没有索引的表时，就会发生全表扫描，这通常会需要较长的时间才能返回数据。对于大多数大型表来说，应该考虑设置索引。而对于小型表来说，就像前面已经说过的，即使表里有索引，优化器也可能会选择全表扫描而不是使用索引。对于具有索引的小型表来说，可以考虑删除索

引，从而释放索引所占据的空间，使其可以用于数据库的其他对象。

提示：简单方法避免全表扫描

除了确保表里存在索引之外，避免全表扫描的最简单、最明显方法是在查询的WHERE子句里设置条件来过滤返回的数据。

下面是应该被索引的数据：

作为主键的字段；

作为外键的字段；

在结合表里经常使用的字段；

经常在查询里作为条件的字段；

大部分值是唯一值的字段。

注意：全表扫描也有好处

有时全表扫描也是好的。对小型表进行的查询，或是会返回表里大部分记录的查询应该执行全表扫描。强制执行全表扫描的最简单方式是不给表创建索引。

## [17.5 其他性能考虑](#)

在调整SQL语句里还有其他一些性能考虑，后面的小节将讨论如下概念：

使用LIKE操作符和通配符；

避免OR操作符；

避免HAVING子句；

避免大规模排序操作；

使用存储过程；

在批加载时关闭索引。

### [17.5.1 使用LIKE操作符和通配符](#)

LIKE 操作符是个很有用的工具，它能够以灵活的方式为查询设置

条件。在查询里使用通配符能够消除很多可能返回的记录。对于搜索类似数据（不等于特定值的数据）的查询来说，通配符是非常灵活的。

假设我们要编写一个查询，从表EMPLOYEE\_TBL里选择字段EMP\_ID、LAST\_NAME、FIRST\_NAME和STATE，获得姓为Stevens的雇员ID、姓名和所在的州。下面3个范例使用了不同的通配符。

第一个查询：

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME, STATE
FROM EMPLOYEE_TBL
WHERE LAST_NAME LIKE 'STEVENS';
```

第二个查询：

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME, STATE
FROM EMPLOYEE_TBL
WHERE LAST_NAME LIKE '%EVENS%';
```

下面是第三个查询：

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME, STATE
FROM EMPLOYEE_TBL
WHERE LAST_NAME LIKE 'ST%';
```

这些SQL语句并不是必须返回同样的结果。更可能的情况是，查询1利用了索引的优势，返回的记录比其他两个查询少。查询2和查询3没有明确指定要返回的数据，其检索速度要比查询1慢。另外，查询3应该比查询2更快，因为它指定了搜索字符串的开头字符（而且字段LAST\_NAME很可能具有索引），因此它能够利用索引。

注意：说明数据存在的差别

查询1可能会返回姓为Stevens的全部雇员，但难道Stevens不能有其他拼写方式了吗？查询2会返回姓为Stevens及其他拼写方式的全部雇员。查询3返回姓以St开头的全部雇员，这是确保获取全部姓

Stevens（或Stephens）的记录的唯一方式。

### [17.5.2 避免使用OR操作符](#)

在SQL语句里用谓词IN代替OR操作符能够提高数据检索速度。SQL实现里有计时工具或其他检查工具，可以反应出OR操作符与谓词IN之间的性能差别。下面的一个范例将展示如何用IN代替OR来重新构造SQL语句。

注意：如何使用**OR**和**IN**

关于OR操作符和谓词IN请参见第8章。

下面是使用OR操作符的查询：

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME
FROM EMPLOYEE_TBL
WHERE CITY = 'INDIANAPOLIS'
       OR CITY = 'BROWNSBURG'
       OR CITY = 'GREENFIELD';
```

下面是同一个查询，使用了谓词IN：

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME
FROM EMPLOYEE_TBL
WHERE CITY IN ('INDIANAPOLIS', 'BROWNSBURG',
              'GREENFIELD');
```

这两个SQL返回完全相同的数据，但通过测试可以发现，用IN代替OR后，检索数据的速度明显提高了。

### [17.5.3 避免使用HAVING子句](#)

HAVING子句是很有用的，可以减少GROUP BY子句返回的数据，但使用它也要付出代价。HAVING子句会让SQL优化器进行额外的工作，也就需要额外的时间。这样的查询既要返回的结果集进行分组，

又要根据HAVING子句的限制条件对结果集进行分析。看下面的例子：

```
SELECT C.CUST_ID, C.CUST_NAME, P.PROD_DESC,  
       SUM(O.QTY) AS QTY, SUM(P.COST) AS COST,  
       SUM(O.QTY * P.COST) AS TOTAL  
FROM CUSTOMER_TBL AS C  
     INNER JOIN ORDERS_TBL AS O ON C.CUST_ID = O.CUST_ID  
     INNER JOIN PRODUCTS_TBL AS P ON O.PROD_ID = P.PROD_ID  
WHERE PROD_DESC LIKE ('P%')  
GROUP BY C.CUST_ID, C.CUST_NAME, P.PROD_DESC  
HAVING SUM(O.QTY * P.COST)>25.00
```

在这个例子中，我们需要找到对某个产品的总计消费超过 25 元的客户。这个查询很简单，而且我们的示例数据库也很小，但HAVING子句的使用仍然增加了额外的工作，尤其当HAVING子句包含了复杂的逻辑而又应用于大量数据的时候。在可能的情况下，尽量不要在SQL语句中使用HAVING子句，如果需要使用，则最好尽可能地使其中的限制条件简单化。

#### [17.5.4 避免大规模排序操作](#)

大规模排序操作意味着使用ORDER BY、GROUP BY和HAVING子句。无论何时执行排序操作，都意味着数据子集必须要保存到内存或磁盘里（当已分配的内存空间不足时）。数据是经常需要排序的，排序的主要问题是会影响SQL语句的响应时间。由于大规模排序操作不是总可以避免的，所以最好把大规模排序在批处理过程里，在数据库使用的非繁忙期运行，从而避免影响大多数用户进程的性能。

#### [17.5.5 使用存储过程](#)

我们可以为经常运行的SQL语句（特别是大型事务或查询）创建存储过程。所谓存储过程就是经过编译的、以可执行格式永久保存在数据库里的SQL语句。

一般情况下，当SQL语句被提交给数据库时，数据库必须检查它的语法，并且把语句转化为可以在数据库里执行的格式（称为解析）。语句被解析之后就保存在内存里，但这并不是持久的。也就是说，当其他操作需要使用内存时，语句就会被从内存里释放。而在使用存储过程时，SQL语句总是处于可执行格式，并且一直会保存在数据库里，直到像别的数据库对象一样被删除。关于存储过程的详细介绍请见第22章。

#### 17.5.6 在批加载时关闭索引

当用户向数据库提交一个事务时（INSERT、UPDATE或DELETE），表和与这个表相关联的索引里都会有数据变化。这意味着如果表EMPLOYEE里有一个索引，而用户更新了表EMPLOYEE，那么相关索引也会被更新。在事务环境里，虽然对表的每次写入都会导致索引也被写入，但一般不会产生什么问题。

然而在批量加载时，索引可能会严重地降低性能。批加载可能包含数百、数千或数百万操作语句或事务，由于规模较大，批加载需要较长的时间才能完成，而且通常安排在非高峰期使用，一般是在周末或夜晚。为了优化批加载的性能——需要12小时完成的批加载可能缩短为6小时——最好在加载过程中关闭相应表的索引。当相应的索引被删除之后，对表所做的修改会在更短的时间内完成，整个操作也会更快地完成。当批加载结果之后，我们可以重建索引。在索引的重建过程中，表里适当的数据会被填充到索引。虽然对于大型表来说，创建索引需要一定的时间，但从整体来看，先删除索引再重建它所需要的时间要更少一些。

在批加载操作的前后删除并重建索引的方法还有另一个优点，就是可以减少索引里的碎片。当数据库不断增长时，记录被添加、删除和更新，就会产生碎片。对于不断增长的数据库来说，最好定期地删除和重建索引。当索引被重建时，构成索引的物理空间数量减少了，也就减少



了读取索引所需的磁盘I/O，用户就会更快地得到结果，皆大欢喜。

## 17.6 基于成本的优化

用户可能经常会遇到需要进行SQL语句调整的数据库。这类系统在任何一个时间点上往往都有数千条SQL语句正在执行。要优化进行调整所花费的时间，需要首先确定需要调整的查询类型。这就是我们所关注的，基于成本的优化试图确定什么样的查询造成了系统资源的额外消耗。例如，如果我们用运行时间来作为衡量标准的话，如下两个查询会获得相应的运行时间：

```
SELECT * FROM CUSTOMER_TBL
WHERE CUST_NAME LIKE '%LE%'                2 sec

SELECT * FROM EMPLOYEE_TBL
WHERE LAST_NAME LIKE 'G%';                 1 sec
```

简单来看，第1条语句似乎就是我们需要进行优化的查询。但是，如果第2条语句每小时执行1000次，而第1条语句每小时仅执行10次，情况又怎么样呢？结果完全相反。

基于成本的优化根据资源消耗量对SQL语句进行排序。根据查询的衡量方法（如执行时间、读库次数等）以及给定时间段内的执行次数，可以方便地确定资源消耗量：

总计资源消耗 = 衡量方法 × 执行次数

使用这种方法，可以最大程度地获得调整收益。在上面的例子中，如果我们能够将每条语句的运行时间减半，就可以很方便地看出所节省的时间：

Statement #1: 1 sec \* 10 executions = 10 sec of computational savings

Statement #2: .5 sec \* 1000 executions = 500 sec of computational savings



这样就很容易理解，为什么要把宝贵的时间花在第2条语句上了。这不仅优化了数据库，也同时优化了用户的时间。

## [17.7 性能工具](#)

很多关系型数据库具有内置的工具用于SQL语句和数据库性能调整。举例来说，Oracle有一个名为EXPLAIN PLAN的工具，可以向用户显示SQL语句的执行计划。还有一个工具是TKPROF，它可以测量SQL语句的实际执行时间。在SQL Server里有一个Query Analyzer，可以向用户提供估计的执行计划或已执行查询的统计参数。关于可以使用的工具请询问DBA或查看相应的文档。

## [17.8 小结](#)

本章介绍了在关系型数据库里调整SQL语句的含义，介绍了两种基本的调整类型：数据库调整和SQL语句调整，它们对于提高语句的执行效率都是很重要的。它们具有同等的重要性，只调整一个无法达到优化目的。

本章介绍了调整SQL语句的方法，首先是语句的可读性，虽然它不能直接改善性能，但有助于程序员开发和管理语句。SQL语句性能中一个重要因素是索引的使用，有时需要使用，&nbsp;有时则需要避免。对于任何用于改善SQL语句性能的方法来说，最重要的是要理解数据本身、数据库设计和关系以及用户的需求。

## [17.9 问与答](#)

问：通过遵循本章所介绍的规则，以数据检索时间来说，在实际应用中能够获得多大的性能提升呢？

答：在实际应用中，检索时间可能缩短几分之一秒，或是几分钟、

几小时，甚至是几天。

问：如何测试**SQL**语句的性能？

答：每个SQL实现都应该有一个工具或系统来测试性能。本书中使用了Oracle7来测试SQL语句，它有多个工具可以测试性能，包括EXPLAIN PLAN、TKPROF和SET命令。每个实现里的具体工具及其使用请参考相应的文档。

## **17.10 实践**

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### **17.10.1 测验**

1. 在小规模表上使用唯一索引会带来什么好处吗？
2. 当执行查询时，如果优化器决定不使用表上的索引，会发生什么呢？
3. WHERE子句里的最严格条件应该放在结合条件之前还是之后呢？

### **17.10.2 练习**

1. 改写下面的SQL语句来改善性能。使用如下所示的表EMPLOYEE\_TBL和表EMPLOYEE\_PAY\_TBL。

```

EMPLOYEE_TBL
EMP_ID          VARCHAR(9)          NOT NULL          Primary key,
LAST_NAME       VARCHAR(15)         NOT NULL,
FIRST_NAME      VARCHAR(15)         NOT NULL,
MIDDLE_NAME     VARCHAR(15),
ADDRESS         VARCHAR(30)         NOT NULL,
CITY            VARCHAR(15)         NOT NULL,
STATE           VARCHAR(2)          NOT NULL,
ZIP             INTEGER(5)          NOT NULL,
PHONE           VARCHAR(10),
PAGER           VARCHAR(10),
CONSTRAINT EMP_PK PRIMARY KEY (EMP_ID)

```

```

EMPLOYEE_PAY_TBL
EMP_ID          VARCHAR(9)          NOT NULL          primary key,
POSITION        VARCHAR(15)         NOT NULL,
DATE_HIRE       DATETIME,
PAY_RATE        DECIMAL(4,2)        NOT NULL,
DATE_LAST_RAISE DATETIME,
SALARY          DECIMAL(8,2),
BONUS          DECIMAL(8,2),
CONSTRAINT EMP_FK FOREIGN KEY (EMP_ID)
REFERENCES EMPLOYEE_TBL (EMP_ID)

```

**a.**

```

SELECT EMP_ID, LAST_NAME, FIRST_NAME,
       PHONE
FROM EMPLOYEE_TBL
WHERE SUBSTRING(PHONE, 1, 3) = '317' OR
      SUBSTRING(PHONE, 1, 3) = '812' OR
      SUBSTRING(PHONE, 1, 3) = '765';

```

**b.**

```

SELECT LAST_NAME, FIRST_NAME
FROM EMPLOYEE_TBL
WHERE LAST_NAME LIKE '%ALL%';

```

c.

```
SELECT E.EMP_ID, E.LAST_NAME, E.FIRST_NAME,
       EP.SALARY
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL EP
WHERE LAST_NAME LIKE 'S%'
      AND E.EMP_ID = EP.EMP_ID;
```

2. 添加一个名为EMPLOYEE\_PAYHIST\_TBL的表，用于存放大量的支付历史数据。使用下面的表来编写SQL语句，解决后续的问题。

```
EMPLOYEE_PAYHIST_TBL
PAYHIST_ID          VARCHAR(9)      NOT NULL      primary key,
EMP_ID              VARCHAR(9)      NOT NULL,
START_DATE          DATETIME        NOT NULL,
END_DATE            DATETIME,
PAY_RATE            DECIMAL(4,2)    NOT NULL,
SALARY              DECIMAL(8,2)    NOT NULL,
BONUS              DECIMAL(8,2)    NOT NULL,
CONSTRAINT EMP_FK FOREIGN KEY (EMP_ID)
REFERENCES EMPLOYEE_TBL (EMP_ID)
```

首先思考，用什么方法能够确定所写的查询可以正确执行？

a. 查询正式员工（salaried employee）和非正式员工（nonsalaried employee）在付薪第一年各自的总人数。

b. 查询正式员工和非正式员工在付薪第一年各自总人数的差异。其中，非正式员工全年无缺勤（ $PAY\_RATE * 52 * 40$ ）。

c. 查询正式员工现在和刚入职时的薪酬差别。同样，非正式员工全年无缺勤。并且，员工的薪水在EMPLOYEE\_PAY\_TBL和EMPLOYEE\_PAYHIST\_TBL两个表中都有记录。在支付历史表中，当前支付记录的END\_DATE字段为NULL值。

## 第六部分 使用SQL管理用户和安全

第18章 管理数据库用户

第19章 管理数据库安全

### 第18章 管理数据库用户

本章的重点包括：

用户的类型

用户管理

用户在数据库的位置

用户与规划

用户会话

修改用户的属性

用户特征

从数据库删除用户

用户使用的工具

本章介绍关系型数据库一个最关键的管理功能：管理数据库用户。该功能可以确保指定用户和应用对数据库的访问，并拒绝非指定的外部访问。考虑到数据库中大量敏感的商业和个人信息，本章的内容绝对是用户需要特别留心掌握的。

#### 18.1 数据库的用户管理

用户是我们所做一切工作的原因：设计、创建、实现和维护数据库。在数据库设计时就考虑了用户的需求，而实现数据库的最终目标是把它交给用户，让用户使用。

关于用户的一个公认理解是，如果没有用户，数据库就不会发生任何不好的事情。虽然这句话貌似真理，但创建数据库就是为了保存数据，从而让用户在每天的工作中使用它们。

虽然用户管理通常是数据库管理员的份内工作，其他人有时也会参与到用户管理过程中。用户管理是关系型数据库生存周期内一件非常重要的工作，它最终是通过使用SQL概念和命令来实现的。对于数据库管理员来说，用户管理的最终目标是在让用户访问所需的数据与保持数据完整性之间寻求平衡。

注意：用户的身份会变化

不同场合的用户的名称、任务、职责之间有很大差别，取决于每个组织的规模和特定的数据处理需求。一个组织的DBA可能是另一个组织里的普通工作人员。

### 18.1.1 用户的类型

数据库用户的类型有多种，包括：

数据输入员；  
程序员；  
系统工程师；  
数据库管理员；  
系统分析员；  
开发人员；  
测试人员；  
管理者；  
终端用户。

每种用户都有其特定的工作职责（和要求），这对他们的每日工作与职位稳定都是很重要的。另外，每种用户在数据库里具有不同的权限级别和自己的位置。

### 18.1.2 谁管理用户

公司的管理人员负责日常的人员管理，而数据库管理员或其他被指定的人负责管理数据库里的用户。

数据库管理员（DBA）通常负责创建数据库用户账户、角色、权限和特征，以及相应的删除操作。在大型实用环境中，这可能是件非常繁重的工作，有些公司会安排一个安全员协助DBA进行用户管理。

这个安全员主要负责一些文书工作，向DBA传递用户的工作需求，让DBA知道哪些用户不再需要访问数据库了。

系统分析员或系统管理员通常负责操作系统安全，包括创建用户和分配适当的权限。安全员可以像帮助数据库管理员一样帮助系统分析员。

以有序的方式分配和撤销权限，并且记录所做的修改，这样可以让管理过程轻松一些。另外，当系统需要进行内部或外部审核时，文档也会提供很好的记录信息。本章将重点介绍用户管理系统。

### 18.1.3 用户在数据库里的位置

用户需要被赋予一定的角色和权限才能完成自己的工作，但用户的权限也不能超出其工作范围。设置用户账户和安全的唯一也是全部原因就是保护数据。如果错误的用户访问了错误的数据，即使是在无意情况下，数据也可能被毁坏或丢失。当用户不再需要访问数据库时，相应的账户应该尽快从数据库里删除或禁止。

注意：确保进行系统的用户管理

用户账户管理对于数据库保护和成功应用是至关重要，如果没有实施有系统的管理，它一般会失败的。从理论上讲，用户账户管理是最简单的数据库管理任务之一，但通常会由于政策因素与通信问题而复杂化。

全部用户在数据库里都有位置，有些具有更多的责任和与众不同的

职责。数据库用户就像是我们身体的各个部分，以一个整体共同作用来达到某些目标。

#### 18.1.4 不同规划里的用户

数据库对象与数据库用户账户相关联，被称为规划。规划是数据库用户拥有的数据库对象集，这个用户被称为规划所有人。规划在逻辑上的组织类似于数据库中的对象，由一个特定的所有人进行管理。例如，可以将所有的人事表组织起来成为一个名为HR的规划，便于进行人力资源管理。普通数据库用户与规划所有人之间的区别在于后者在数据库里拥有对象，而大多数用户没有自己的对象，只是被赋予数据库账户来访问规划里的数据。由于规划所有人实际上拥有这些对象，所以对它们有完全的控制。

Microsoft SQL Server中进一步设置了数据库所有人。数据库所有人拥有数据库中的所有对象，并且对其中存储的数据拥有完全控制。数据库中有一个或多个规划。数据库以及数据库所有人的默认规划，一般是dbo。可以根据需要，对数据库中的对象进行组织形成多个规划，并指定规划所有人。

注意：不同系统中用户的创建和管理也不同

关于创建用户的实际操作请查看具体实现的帮助文档。在创建和管理用户时，还要遵守公司政策和手续。下面的小节介绍了在Oracle、MySQL、Sybase和Microsoft SQL Server里创建用户的操作。

### 18.2 管理过程

在任何数据库系统里，一个稳定的用户管理系统对于数据安全来说是必不可少的。用户管理系统从新用户的直接上级开始，他负责发起访问请求，然后通过公司的批准程序。如果管理层接受了请求，就会转到安全员或数据库管理员来完成实际操作。一个好的通知过程是必要



的，在用户账户被创建、对数据库的访问被批准之后，管理人和用户必须得到通知，用户账户的密码应该只交给用户本人，而他应该在第一次登录到数据库后就立即修改密码。

### 18.2.1 创建用户

创建数据库用户需要使用数据库里的SQL命令，但并不存在着什么标准命令，每个实现都有自己的方法。不同实现中的基本概念都是一样的。另外还有一些图形化用户界面（GUI）工具可以进行用户管理。

当 DBA 或指定的安全员收到用户账户请求时，应该针对必要信息进行分析。这些信息应该包含公司对于创建用户ID所必需的条件。

一些必要信息包括社会保险号码、完整姓名、地址、电话号码、办公室或部分名称、被分配的数据库，有时还可以包括建议使用的用户名。

下面的小节将展示在不同实现里创建用户的范例。

#### 一、在**Oracle**里创建用户

下面是在Oracle数据库里创建用户账户的步骤。

1. 使用默认设置创建数据库用户账户。
2. 给用户账户授予适当的权限。

下面是创建用户的语法：

```
CREATE USER USER_ID
IDENTIFIED BY [PASSWORD | EXTERNALLY ]
[ DEFAULT TABLESPACE TABLESPACE_NAME ]
[ TEMPORARY TABLESPACE TABLESPACE_NAME ]
[ QUOTA (INTEGER (K | M) | UNLIMITED) ON TABLESPACE_NAME ]
[ PROFILE PROFILE_TYPE ]
[PASSWORD EXPIRE | ACCOUNT [LOCK | UNLOCK]
```

如果不是在使用 Oracle，我们不必过于关注其中的选项。

Tablespace（表空间）是容纳数据库对象（比如表和索引）的逻辑区

域，是由DBA管理的。DEFAULT TABLESPACE指定用户在创建对象时所在的表空间，而TEMPORARY TABLESPACE是用于排序操作（表结合、ORDER BY、GROUP BY）的表空间。QUOTA是被限制在用户所访问的特定表空间上的空间，而PROFILE是指派给用户的数据库特征文件。

下面是给用户账户授予权限的语法：

```
GRANT PRIV1 [ , PRIV2, ... ] TO USERNAME | ROLE [ , USERNAME ]
```

注意：**CREATE USER**命令也有差别

上面的语法可以向 Oracle 数据库和其他一些主流关系型数据库添加用户。MySQL不支持CREATE USER命令，它使用mysqladmin工具管理用户。在Windows计算机上创建了一个本地用户账户之后，并不需要登录，但在多用户环境里，每个要访问数据库的用户都要创建一个账户。

GRANT 语句可以在同一个语句里给一个或多个用户授予一个或多个权限。权限也可以授予一个角色，然后再授予用户。

在MySQL里，GRANT命令可以把本地计算机上的用户授权到当前数据库里，比如：

```
GRANT USAGE ON *.* TO USER@LOCALHOST IDENTIFIED BY 'PASSWORD';
```

像下面这样给用户授予其他权限：

```
GRANT SELECT ON TABLENAME TO USER@LOCALHOST;
```

在大多数情况下，只有在多用户环境下才需要设置MySQL的多用户。

## 二、在Microsoft SQL Server里创建用户

在Microsoft SQL Server里创建用户账户的步骤如下所述。

1. 为SQL Server创建登录账户，指定密码和默认的数据库。

2. 把用户添加到适当的数据库，从而创建一个数据库账户。
3. 给数据库账户分配适当的权限。

下面是创建用户账户的语法：

```
SP_ADDLOGIN USER_ID ,PASSWORD [ , DEFAULT_DATABASE ]
```

注意：更多的权限内容

第19章将更详细地介绍关系型数据库里的权限问题。

下面是把用户添加到数据库的语法：

```
SP_ADDUSER USER_ID [ , NAME_IN_DB [ , GRPNAME ] ]
```

从上述内容可以看出，SQL Server将登录账户和数据库账户区别对待，登录账户用于访问SQL Server实例，而数据库账户则可以访问数据库对象。创建好登录账户后，在数据库级别运行SP\_ADDUSER命令后，就可以在SQL Server Management Studio的安全文件夹中看到二者的区别。这是SQL Server的一个重要特点，你可以创建一个登录账户，但却不能用这个账户访问实例中的任何数据库。

在SQL Server中创建账户的一个常见错误，就是忘记为账户授权访问其默认数据库。所以在设置账户的时候，务必确保为账户授权，至少保证其能够访问默认数据库，否则在使用账户登录系统的时候就会报错。

下面是给用户账户分配权限的语法：

```
GRANT PRIV1 [ , PRIV2, ... ] TO USER_ID
```

### 三、在MySQL里创建用户

在MySQL里创建用户账户的步骤如下所述。

1. 在数据库里创建用户账户。
2. 给用户账户分配适当的权限。

创建用户账户的语法与Oracle的很类似：

```
SELECT USER user [ IDENTIFIED BY [ PASSWORD ] 'password' ]
```

分配用户权限的语法也与Oracle很相似：

```
GRANT priv_type [(column_list)] [, priv_type [(column_list)] ] ...  
  ON [object_type]  
    {tbl_name | * | *.* | db_name.* | db_name.routine_name}  
  TO user
```

### [18.2.2 创建规划](#)

规划是使用CREATE SCHEMA语句创建的。

其语法如下所示：

```
CREATE SCHEMA [ SCHEMA_NAME ] [ USER_ID ]  
              [ DEFAULT CHARACTER SET CHARACTER_SET ]  
              [ PATH SCHEMA NAME [, SCHEMA NAME] ]  
              [ SCHEMA_ELEMENT_LIST ]
```

下面是一个范例：

```
CREATE SCHEMA USER1  
CREATE TABLE TBL1  
  (COLUMN1    DATATYPE    [NOT NULL],  
   COLUMN2    DATATYPE    [NOT NULL]...)  
CREATE TABLE TBL2  
  (COLUMN1    DATATYPE    [NOT NULL],  
   COLUMN2    DATATYPE    [NOT NULL]...)  
GRANT SELECT ON TBL1 TO USER2  
GRANT SELECT ON TBL2 TO USER2  
[ OTHER DDL COMMANDS ... ]
```

下面是在一个实现里使用CREATE SCHEMA命令的实例：

```

CREATE SCHEMA AUTHORIZATION USER1
CREATE TABLE EMP
  (ID      NUMBER      NOT NULL,
   NAME    VARCHAR2(10) NOT NULL)
CREATE TABLE CUST
  (ID      NUMBER      NOT NULL,
   NAME    VARCHAR2(10) NOT NULL)
GRANT SELECT ON TBL1 TO USER2
GRANT SELECT ON TBL2 TO USER2;
Schema created.

```

这个命令里添加了关键字 **AUTHORIZATION**，它是在 Oracle 数据库里执行的。从这个范例以及前面的很多范例中都可以看出，不同实现的命令语法有所不同。

能够创建规划的实现会为用户分配一个默认规划，该规划通常与用户的账户相关联。所以，如果一个用户的账户名为 **BethA2**，那么他的默认规划名通常就为 **BethA2**。这一点很重要，如果在创建对象的时候不指定规划名，那么将在用户的默认规划中创建对象。如果我们在 **BethA2** 账户中运行下面的 **CREATE TABLE** 语句，将在 **BethA2** 默认规划中创建表：

```

CREATE TABLE MYTABLE(
  NAME VARCHAR(50) NOT NULL );

```

但这里有可能并不是用户所希望创建表的位置。如果是在 **SQL Server** 中，我们拥有 **dbo** 规划的访问权限，并且要在该规划中创建表。这时，需要对所创建的对象进行如下限定：

```

CREATE TABLE DBO.MYTABLE(
  NAME VARCHAR(50) NOT NULL):

```

在创建用户账户并分配权限的时候，务必牢记上述问题。这样可以确保在用户的数据库中维持一个恰当的秩序，以避免不良后果。

### [18.2.3 删除规划](#)

使用DROP SCHEMA语句可以从数据库里删除规划，这时必须要考虑两个选项。一个是RESTRICT，在使用这个选项时，如果规划里有对象，删除操作就会发生错误。第二个选项是 CASCADE，如果规划里有对象，删除规划就必须指定这个选项。记住，当我们删除规划时，与规划相关联的全部数据库对象都会被删除。

注意：不是所有实现都支持**CREATE SCHEMA**命令

有些实现可能不支持CREATE SCHEMA命令，但当用户创建对象时会隐含地创建规划，而CREATE SCHEMA命令只不过是完成这个任务的一个单步方法而已。当用户创建对象之后，可以向其他用户分配访问这些对象的权限。MySQL不支持CREATE SCHEMA命令。在MySQL里，规划被看作一个数据库，所以我们要使用CREATE DATABASE命令来创建一个规划，然后在其中创建对象。

其语法如下所示：

```
DROP SCHEMA SCHEMA_NAME { RESTRICT | CASCADE }
```

注意：删除规划的不同方法

如果发现规划里缺少了某些对象，很可能是由于对象（比如表）会被像DROP TABLE 这样的命令删除。有些实现提供了删除用户的过程或命令，也可以用于删除规划。如果所使用的SQL实现里没有DROP SCHEMA命令，我们可以通过删除拥有规划对象的用户来删除规划。

### [18.2.4 调整用户](#)

用户管理中的一个重要组成部分是在创建用户之后修改用户的属性。如果具有用户账户的个人永远不会升职、不会离开公司，或者新雇员非常少，DBA的工作就会轻松很多。但在现实世界里，频繁的人员调

动和职责变化是用户管理中的重要因素，几乎每个人都会改变工作或职责。因此，数据库的用户权限必须进行相应的调整以适应用户的需要。

下面是Oracle里修改用户状态的范例：

```
ALTER USER USER_ID [ IDENTIFIED BY PASSWORD | EXTERNALLY | GLOBALLY AS  
'CN=USER']  
[ DEFAULT TABLESPACE TABLESPACE_NAME ]  
[ TEMPORARY TABLESPACE TABLESPACE_NAME ]  
[ QUOTA INTEGER K|M |UNLIMITED ON TABLESPACE_NAME ]  
[ PROFILE PROFILE_NAME ]  
[ PASSWORD EXPIRE]  
[ ACCOUNT [LOCK |UNLOCK]]  
[ DEFAULT ROLE ROLE1 [, ROLE2 ] | ALL  
[ EXCEPT ROLE1 [, ROLE2 | NONE ] ]
```

这个语句可以改变用户的很多属性，但并不是所有SQL实现都提供了这样一个简单的命令来操作数据库用户。

比如 MySQL，它使用多种手段来调整用户账户。举例来说，使用如下语法重置用户的密码：

```
UPDATE mysql.user SET Password=PASSWORD('new password')  
WHERE user='username';
```

而使用下面的语法来改变用户的用户名：

```
RENAME USER old_username TO new_username;
```

有些实现还提供了GUI工具来创建、修改和删除用户。

注意：在数据库和工具中不使用手工输入命令

记住，不同实现中的语法是不一样的。另外，大多数数据库用户不会手工向数据库发出连接和断开的命令，而是使用厂商提供的工具或第三方工具来输入用户名和密码，从而连接到数据库并初始化数据库用户会话。

### [18.2.5 用户会话](#)

一个用户数据库会话就是从登录数据库到退出这段时间。在一个用户会话中，用户可以执行允许范围内的各种操作，比如查询和事务。

基于创建的连接和会话，用户可以执行任意数量的事务，直到连接中断，这时数据库用户会话也结束了。

使用下面这样的命令可以明确地连接和断开数据库，从而开始和结束SQL会话：

```
CONNECT TO DEFAULT | STRING1 [ AS STRING2 ] [ USER STRING3 ]  
DISCONNECT DEFAULT | CURRENT | ALL | STRING  
SET CONNECTION DEFAULT | STRING
```

用户会话能够——并且经常——被 DBA 或其他对用户行为感兴趣的人监视。用户会话是与特定用户相关联的。在主机的操作系统上，数据库用户会话实际上是一个进程。

### [18.2.6 禁止用户访问](#)

通过几个简单的命令就可以从数据库里删除用户或禁止用户的访问，但在不同实现里具体的命令依然是不一样的，请查看相应的文档来了解实际的语法或工具。

下面是禁止用户访问数据库的一些方法：

修改用户的密码；

从数据库删除用户账户；

撤销分配给用户的相应权限。

有些实现里可以使用DROP命令删除数据库里的用户：

```
DROP USER USER_ID [ CASCADE ]
```

在很多实现里，与GRANT命令执行相反操作的是REVOKE，用于



取消已经分配给用户的权限。这个命令在SQL Server、Oracle和MySQL里的语法如下所示：

```
REVOKE PRIV1 [ ,PRIV2, ... ] FROM USERNAME
```

### [18.3 数据库用户使用的工具](#)

有些人认为不必了解SQL就可以执行数据库查询，这在有些情况下是正确的。然而即使是在使用GUI工具时，了解SQL也绝对会对查询操作有所帮助。GUI工具很不错，在方便得到时也应该使用它们，但理解其幕后的工作原理对于最有效地利用这些用户友好的工具也大有益处。

很多GUI工具帮助数据库用户自动生成SQL代码，用户只需要在一些窗口里浏览、对一些提示做出响应、选择一些选项即可。还有专门生成报告的工具，还可以为用户创建窗口来查询、更新、插入或删除数据库里的数据。有一些工具可以把数据转化为图形或图表，还有数据库管理工具可以监视数据库性能，有些还可以远程连接到数据库。数据库厂商提供了其中一部分工具，其他的工具则来自于第三方厂商。

### [18.4 小结](#)

所有的数据库都有用户，无论是只有一个，还是成千上万。用户是数据库存在的原因。

用户管理有3个基本要素。首先，必须能够为特定的人和服务创建数据库用户账户。其次，必须能够为用户账户分配权限，使其能够完成要对数据库所做的操作。最后，必须能够从数据库里删除用户账户，或是撤销相应的权限。

本章介绍了用户管理中最常见的任务，但没有涉及过多的细节，因为大多数数据库在用户管理过程上是不同的。但由于用户管理与SQL的关系，在此对其进行讨论还是必要的。很多用于管理用户的命令在

ANSI标准没有定义或详细讨论，但概念还是相同的。

## [18.5 问与答](#)

问：向数据库添加用户有什么**SQL**标准吗？

答：ANSI 提供了一些命令和概念，但在创建用户方面每种实现和每家公司都有自己的命令、工具和规则。

问：在不把用户**ID**从数据库里彻底删除的情况下，有没有办法暂时禁止用户的访问？

答：有。要想暂时禁止用户的访问，只需要改变用户的密码，或是撤销允许用户连接到数据库的权限。之后，如果想恢复用户账户的功能，只需要把修改的密码告诉用户，或是分配适当的权限。

问：用户能改变自己的密码吗？

答：在大多数主流实现里是可以的。在创建用户或把用户添加到数据库的过程中，一般会为用户设置一个普通的密码，而且必须尽快由用户修改为自己所选择的密码。密码修改之后，即使**DBA**也不知道用户的密码。

## [18.6 实践](#)

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### [18.6.1 测验](#)

1. 使用什么命令创建会话？
2. 在删除包含数据库对象的规划时，必须要使用什么选项？

3. MySQL里使用什么命令创建规划？
4. 使用什么命令清除数据库权限？
5. 什么命令能够创建表、视图和权限的组或集合？
6. 在SQL Server中，登录账户和数据库账户有什么区别？

#### 18.6.2 练习

1. 描述如何在learnsql数据库里创建一个新用户“John”。
2. 如何让新用户John能够访问表Employee\_tbl？
3. 描述如何设置John的权限，让他访问learnsql数据库里的全部对象。
4. 描述如何撤销John的权限，然后删除他的账户。

### 第19章 管理数据库安全

本章的重点包括：

数据库安全

安全与用户管理

数据库系统权限

数据库对象权限

给用户分配权限

撤销用户的权限

数据库里的安全特征

本章介绍使用SQL命令和相关命令在关系型数据库里实现和管理安全的基本知识。不同主流实现的安全命令在语法上是有区别的，但关系型数据库的整体安全概念都遵循 ANSI 标准。关于安全操作的详细语法和方针请查看具体实现的文档。

## 19.1 什么是数据库安全

数据库安全就是保护数据不受到未授权访问。那些只能对数据库里部分数据进行访问的用户，如果要访问其他的数据，也属于未授权访问。这种保护还包括防止未授权的连接和权限分配。数据库里存在多个用户级别，从数据库创建者，到负责维护数据库的人员（比如数据库管理员），到数据库程序员，到终端用户。终端用户虽然在访问权限上受到最大的限制，但却是数据库存在的原因。每个用户对数据库具有不同的访问级别，应该被限制到能够完成相应工作所需的最小权限。

那么，用户管理与数据库安全有什么区别呢？前一章介绍的用户管理似乎涵盖了安全问题。虽然用户管理和数据库安全有着必然的联系，但各自具有不同的目标，共同完成保护数据库的任务。

良好规划和维护的用户管理与数据库的整体安全是密切相关的。用户被分配一定的账户和密码，从而可以对数据库进行一般的访问。数据库里的用户账户应该保存一些用户信息，比如用户的实际姓名、所在的办公室和部门、电话号码或分机号、可以访问的数据库名称。个人用户信息应该只能由DBA访问。新用户一般会由DBA或安全员授予一个初始密码，他在首次登录后应该立即修改这个密码。记住，DBA不需要、也不应该知道个人的密码，这确保了职责的分离，也让用户账户的不断增加不会成为问题。

如果用户不再需要一定的权限，那么这些权限就应该被撤销。如果用户不再需要访问数据库，用户账户就应该从数据库里删除。

一般来说，用户管理是创建用户账户、删除用户账户、跟踪用户在数据库里行为的过程。而数据库安全更进一步，包括了为特定数据库访问授予权限、从用户撤销权限、采取手段保护数据库的其他部分（比如底层数据库文件）。

注意：数据库安全有更多的内容需要学习

由于这是一本SQL图书而不是数据库图书，所以重点在于数据库权限。但我们也要考虑到数据库安全的其他方面，比如保护底层数据库文件，这与数据库权限的配置具有同等的重要性。高级数据库安全可能相当复杂，而且在不同关系型数据库实现里也有所区别。如果想更详细地了解数据库安全，可以查看互联网安全中心的网页。

## [19.2 什么是权限](#)

权限是用于访问数据库本身、访问数据库里的对象、操作数据库里的数据、在数据库里执行各种管理功能的许可级别。权限是通过GRANT命令分配的，用REVOKE命令撤销。

用户可以连接到数据库并不意味着可以访问数据库里的数据，要访问数据库里的数据还需要权限。权限有两种类型，一种是系统权限，一种是对对象权限。

### [19.2.1 系统权限](#)

系统权限允许用户在数据库里执行管理操作，比如创建数据库、删除数据库、创建用户账户、删除用户、删除和修改数据库对象、修改对象的状态、修改数据库的状态以及其他会对数据库造成重要影响的操作。

系统权限在不同关系型数据库实现里差别很大，所以具体权限及其正确用法请查看实现的文档。

下面是SQL Server里一些常见系统权限：

CREATE DATABASE——允许创建新的数据库；

CREATE PROCEDURE——允许创建新的存储过程；

CREATE VIEW——允许创建新的视图；

BACKUP DATABASE——允许用户对数据库进行备份；

CREATE TABLE——允许用户创建新表；

CREATE TRIGGER——允许用户在表上创建触发器；

EXECUTE——允许用户在特定数据库中运行给定的存储过程。

下面是Oracle里一些常见系统权限：

CREATE TABLE——允许用户在特定规划中创建新表；

CREATE ANY TABLE——允许用户在任意规划中创建新表；

ALTER ANY TABLE——允许用户在任意规划中修改表结构；

DROP TABLE——允许用户在特定规划中删除表对象；

CREATE USER——允许用户创建其他用户账户；

DROP USER——允许用户删除既有用户账户；

ALTER USER——允许用户修改既有用户账户；

ALTER DATABASE——允许用户修改数据库特性；

BACKUP ANY TABLE——允许用户备份任意规划中任意表的数据；

SELECT ANY TABLE——允许用户查询任意规划中任意表的数据。

下面是MySQL里一些常见的全局（系统）权限：

CREATE——允许用户创建特定对象，如数据库、表或索引；

DROP——允许用户删除特定对象；

GRANT——允许用户对特定对象分配权限；

RELOAD——允许用户进行清除缓存操作，以便清除缓存中的日志文件等内容；

SHUTDOWN——允许用户关闭MySQL实例。

注意：权限的不同级别

MySQL具有全局权限和对象权限。全局权限类似于系统权限，负责用户对全部数据库对象的访问。

### [19.2.2 对象权限](#)



对象权限是针对对象的许可级别，意味着必须具有适当的权限才能对数据库对象进行操作。举例来说，为了从其他用户的表里选择数据，我们必须首先得到另一个用户的许可。对象权限由对象的所有者授予数据库里的其他用户。记住，这个所有者也被称为规划所有者。

ANSI标准里包含下述对象权限。

**USAGE:** 批准使用指定的域。

**SELECT:** 允许访问指定的表。

**INSERT(column\_name):** 允许对数据插入到指定表的指定字段。

**INSERT:** 允许对数据插入到指定表的全部字段。

**UPDATE(column\_name):** 允许对指定表里的指定字段进行更新。

**UPDATE:** 允许对指定表里的全部字段进行更新。

**REFERENCES(column\_name):** 允许在完整性约束里引用指定表里的指定字段，任何完整性约束都需要这个权限。

**REFERENCES:** 允许引用指定表里的全部字段。

**注意:** 自动授予的权限

对象的所有者自动被授予与对象相关的全部权限。有些SQL实现里还可以利用GRANT OPTION命令分配这些权限，这是一个相当不错的功能，稍后将详细讨论。

大多数SQL实现都遵循这个对象权限列表来控制对数据库对象的访问。

这些对象级别的权限应该用于许可和限制对规划内的对象的访问，可以保护一个规划里的对象不被能够访问其他规划的用户访问。

不同SQL实现里还有其他一些对象权限在此并没有列出来。比如删除其他用户的对象里的数据。关于全部可用的对象级权限，请查看具体实现的文档。

### **19.2.3 谁负责授予和撤销权限**

使用GRANT和REVOKE命令的人通常是DBA，但如果存在着安全管理员，他也有这样的权力。具体要授予和撤销的权限来自于管理层，而且最好进行细致的跟踪，以确保只有被认可的用户才能具有相应的权限。

对象的所有者负责向数据库里的其他用户授予权限。即使 DBA 也不能给数据库用户授予不属于他的对象的权限，虽然有方法可以绕过这种限制。

### [19.3 控制用户访问](#)

用户访问主要是通过用户账户和密码进行控制的，但在大多数主流实现里，这是不足以访问数据库的。创建用户账户只是允许和控制数据库访问的第一步。

在创建了数据库账户之后，数据库管理员、安全官员或某个指定的人必须能够向需要进行数据库操作的用户授予适当的系统级权限，比如创建表或选择表。接下来，规划所有者需要向用户授予访问规划中对象的权限。

SQL 里用两个命令控制数据库访问，包括权限的授予与撤销，分别是 GRANT 和REVOKE。

#### [19.3.1 GRANT命令](#)

GRANT命令用于向现有数据库用户账户授予系统级和对象级权限。

其语法如下所示：

```
GRANT PRIVILEGE1 [, PRIVILEGE2 ] [ ON OBJECT ]  
TO USERNAME [ WITH GRANT OPTION | ADMIN OPTION]
```

下面就是向用户授予一个权限：



```
GRANT SELECT ON EMPLOYEE_TBL TO USER1;  
Grant succeeded.
```

像下面这样给一个用户授予多个权限：

```
GRANT SELECT, INSERT ON EMPLOYEE_TBL TO USER1;  
Grant succeeded.
```

注意到在一个语句里向一个用户授予多个权限时，每个权限是以逗号分隔的。

注意：注意反馈信息

注意提示信息“Grant succeeded”，它表示授权语句成功完成了。这是Oracle的反馈信息。大多数SQL都会提供某种反馈，但所使用的短语不一定相同。

像下面这样给多个用户授予权限：

```
GRANT SELECT, INSERT ON EMPLOYEE_TBL TO USER1, USER2;  
Grant succeeded.
```

## 一、GRANT OPTION

GRANT OPTION是个功能强大的GRANT选项。当对象的所有者利用GRANT OPTION把自己对象的权限授予另一个用户时，这个用户还可以把这个对象的权限授予其他用户，尽管他并不是这个对象的所有者。范例如下：

```
GRANT SELECT ON EMPLOYEE_TBL TO USER1 WITH GRANT OPTION;  
Grant succeeded.
```

## 二、ADMIN OPTION

使用ADMIN OPTION授予权限之后，用户不仅拥有了权限，也具有了把这个权限授予其他用户的能力，这一点与GRANT OPTION类

似。但GRANT OPTION用于对象级权限，而ADMIN OPTION用于系统级权限。当一个用户用ADMIN OPTION向另一个用户授予系统权限之后，后者还可以把系统权限授予其他用户。范例如下：

```
GRANT CREATE TABLE TO USER1 WITH ADMIN OPTION;  
Grant succeeded.
```

### 19.3.2 REVOKE命令

注意：删除用户的同时也删除了权限

当一个被使用GRANT OPTION或ADMIN OPTION授予了权限的用户被删除之后，权限与用户之间的关联也被断开了。

REVOKE命令撤销已经分配给用户的权限，它有两个选项：RESTRICT和CASCADE。当使用RESTRICT选项时，只有当REVOKE命令里指定的权限撤销之后不会导致其他用户产生报废权限时，REVOKE才能顺利完成。而CASCADE会撤销权限，不会遗留其他用户的权限。换句话说，如果某个对象的所有者使用GRANT OPTION把权限授予USER1，USER1又使用了GRANT OPTION向USER2授予权限，然后对象所有者撤销了USER1的权限，这时如果使用CASCADE选项，那么USER2的权限也会被撤销。

当用户使用GRANT OPTION向其他用户授予权限之后，自己被从数据库里删除了，或是自己的权限被撤销了，那么后一个用户的权限就被称为报废权限。

REVOKE命令的语法如下所示：

```
REVOKE PRIVILEGE1 [, PRIVILEGE2 ] [ GRANT OPTION FOR ] ON OBJECT  
FROM USER { RESTRICT | CASCADE }
```

下面是一个范例：

```
REVOKE INSERT ON EMPLOYEE_TBL FROM USER1;  
Revoke succeeded.
```

### [19.3.3 控制对单独字段的访问](#)

我们不仅能够把表作为一个整体来分配对象权限（INSERT、UPDATE和DELETE），还可以分配表里指定字段的权限来限制用户的访问，如下所示：

```
GRANT UPDATE (NAME) ON EMPLOYEES TO PUBLIC;  
Grant succeeded.
```

### [19.3.4 数据库账户PUBLIC](#)

数据库账户PUBLIC是个代表数据库里全体用户的账户。所有用户都属于PUBLIC账户。如果某个权限被授予PUBLIC账户，那么数据库全部用户都具有这个权限。类似地，如果一个权限从PUBLIC上撤销，就相当于从全部数据库用户上撤销了这个权限，除非这个权限明确地授予了特定用户。范例如下：

```
GRANT SELECT ON EMPLOYEE_TBL TO PUBLIC;  
Grant succeeded.
```

### [19.3.5 权限组](#)

有些实现可以在数据库里形成权限组。这些权限组是通过不同的名称来引用的。通过使用权限组，我们可以更方便地给用户授予和撤销权限。举例来说，如果某个权限组具有 10个权限，我们就可以把这个组授予一个用户，而不必授予10个权限。

权限组在Oracle里称为角色。Oracle在其实现里包含以下权限组：

CONNECT——允许用户连接数据库，并且对已经访问过的任何数据库对象进行操作。

注意：不同系统的权限组有所不同

在使用数据库权限组方面，各个实现都有所不同。如果实现支持这个特性，我们可以利用它来减轻数据库安全管理工作。

**RESOURCE**——允许用户创建对象、删除其所拥有的对象、为其所拥有的对象赋予权限等；

**DBA**——允许用户在数据库中对任何对象进行任何操作。

警告：对**PUBLIC**授予的权限可能带来意想不到的后果

向 **PUBLIC** 授予权限时要特别小心。数据库的所有用户都会拥有 **PUBLIC** 的权限，因此在向 **PUBLIC** 授予权限时，可能会意外地让用户能够访问本不该访问的数据。举例来说，如果让 **PUBLIC** 能够从雇员薪水表里选择数据，那么所有用户就可以访问数据库来了解公司发给每个人的工资。

**CONNECT**组允许用户连接到数据库，并且对能够访问的数据库对象执行操作。

**RESOURCE**组允许用户创建对象、删除他拥有的对象、授予他拥有的对象的权限等。

**DBA**组允许用户在数据库里执行任何操作，用户可以访问任何数据库对象，执行任何操作。

把权限组授予用户的范例如下：

```
GRANT DBA TO USER1;  
Grant succeeded.
```

SQL Server在服务器级别和数据库级别有一些权限组。

部分数据库权限组如下：

**DB\_DDLADMIN**

**DB\_DATAREADER**

**DB\_DATAWRITER**

DB\_DDLADMIN角色允许用户使用任意合法的DDL命令，对数据库中的任意对象进行操作。DB\_DATAREADER 角色允许用户在已经获得权限的数据库中，对任意表进行查询。DB\_DATAWRITER角色允许用户对数据库中的任意表运行任何数据控制命令，如INSERT、UPDATE或者DELETE。

## [19.4 通过角色控制权限](#)

角色是数据库里的一个对象，具有类似权限组的特性。通过使用角色，我们不必明确地直接给用户授予权限，从而减少安全维护工作。使用角色可以更方便地进行组权限管理。角色的权限可以被修改，而这种修改对于用户来说是透明的。

如果某个用户需要在一个程序里、在指定时间内对某个表具有SELECT 和 UPDATE 权限，我们可以暂时指派一个具有这些权限的角色，直到事务结束。

当一个角色最初被创建时，它就是数据库里的一个角色，没有任何实际值。角色可以被指派给用户或其他角色。假设在名为 APP01 的规划里，我们把对表 EMPLOYEE\_PAY 的SELECT权限授予角色 RECORDS\_CLERK，那么任何被指派了RECORDS\_CLERK角色的用户或角色就对表EMPLOYEE\_PAY具有了SELECT权限。

类似地，如果APP01从RECORDS\_CLERK角色撤销了对表 EMPLOYEE\_PAY的SELECT权限，任何被指派了RECORD\_CLERK的用户或角色就不再对这个表有SELECT权限了。

在数据库中分配权限的时候，需要考虑好一个用户需要哪些权限，以及其他用户是否需要同样的权限。例如，会计部门的员工需要访问与会计相关的表。在这种情况下，除非这些员工有完全不同的权限要求，否则就可以创建一个角色并为其赋予适当的权限，并将这个角色分配给

这些员工。

如果现在创建了一个新的对象，并需要将相应权限赋给会计部门，就可以方便地在一个位置进行修改，而不必对每一个账户都重新设置。同样，如果会计部门有了一个新成员，或者需要对其他员工赋予同样的权限，只要赋予其相应的角色就可以了。角色是个非常优秀的工具，可以帮助DBA智能地完成工作，即使处理复杂的数据库安全协议也并不麻烦。

### [19.4.1 CREATE ROLE语句](#)

角色是由CREATE ROLE语句创建的：

```
CREATE ROLE role_name;
```

向角色授予权限与向用户授予权限是一样的，范例如下：

```
CREATE ROLE RECORDS_CLERK;  
Role created.  
GRANT SELECT, INSERT, UPDATE, DELETE ON EMPLOYEE_PAY TO RECORDS_CLERK;  
Grant succeeded.  
GRANT RECORDS_CLERK TO USER1;  
Grant succeeded.
```

### [19.4.2 DROP ROLE语句](#)

这个语句用于删除角色

```
DROP ROLE role_name;
```

范例如下：

```
DROP ROLE RECORDS_CLERK;  
Role dropped.
```

注意：**MySQL**不支持角色

MySQL不支持角色。在某些SQL实现里，不支持角色是它们的弱点之一。

### [19.4.3 SET ROLE语句](#)

使用SET ROLE语句可以为用户的SQL会话设置角色：

```
SET ROLE role_name;
```

范例如下：

```
SET ROLE RECORDS_CLERK;  
Role set.
```

一个语句里可以设置多个角色：

```
SET ROLE RECORDS_CLERK, ROLE2, ROLE3;  
Role set.
```

注意：**SET ROLE**语句并不经常使用

在某些实现里，例如Microsoft SQL Server和Oracle，被指派给用户的全部角色都自动成为默认角色。也就是说，只要用户登录到数据库，这些角色就会被设置给用户并发挥作用。这里所介绍的SET ROLE语句，只是为了帮助读者理解相应的ANSI标准。

## [19.5 小结](#)

本章介绍了在SQL数据库或关系型数据库里实现安全的基本知识，包括管理数据库用户的基本方法。在数据库级别为用户实现安全的第一步是创建用户，第二步是为用户分配适当的权限，允许其访问数据库的特定部分。另外，ANSI允许使用角色。权限可以被授予用户或角色。

权限有两种类型，分别是系统权限和对象权限。系统权限允许用户

在数据库里执行各种任务，比如连接到数据库、创建表、创建用户、改变数据库的状态等。对象权限允许用户访问数据库里的指定对象，比如从指定表里选择数据或操作数据。

SQL里有两个命令用于授予和撤销用户或角色的权限：**GRANT**和**REVOKE**。它们用于控制数据库里的整体管理权限。虽然在实现关系型数据库的安全时还需要考虑其他很多因素，但与SQL语言相关的基本内容在此都已经介绍了。

## **19.6 问与答**

问：如果用户忘记了密码，应该如何做才能继续访问数据库呢？

答：用户应该去找直接上级或能够重置用户密码的人员。如果不存在这样的专门人员，**DBA**或安全员可以重置密码。当密码重置之后，用户应该尽快修改为自己的密码。有时**DBA**可以设置一个选项，强制用户在下次登录后立即修改密码。详细情况请参见具体实现的文档。

问：如果想授予某个用户**CONNECT**角色，但该用户不需要**CONNECT**角色的全部权限，应该怎么办？

答：这时不应该授予用户**CONNECT**角色，而是只分配必要的权限。如果已经授予了用户**CONNECT**角色，而用户不在需要这个角色的全部权限，我们就要从用户撤销**CONNECT**角色，然后再分配特定的权限。

问：为什么当新用户从管理者获得新密码之后，立即修改密码是非常重要的？

答：初始密码是根据用户ID设置的。包括**DBA**和管理人员在内的任何人都不应该知道个人的密码。密码应该始终高度保密，从而避免其他用户假冒他人身份登录到数据库。

## **19.7 实践**



下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### 19.7.1 测验

1. 如果用户要把不是其所拥有的对象的权限授予另一个用户，必须使用什么选项？
2. 当权限被授予PUBLIC之后，是数据库的全部用户，还是仅特定用户获得这些权限？
3. 查看指定表里的数据需要什么权限。
4. SELECT是什么类型的权限？
5. 如果想撤销用户对某个对象的权限，以及其他使用GRANT分配这个对象权限的其他用户的权限，应该使用什么选项？

### 19.7.2 练习

1. 登录到你的数据库实例，如果learnsql数据库不是默认的，则转换到learnsql数据库。
2. 在数据库提示符下，根据你所使用的数据库实例，输入相应命令来列出数据库实现中所默认的表：

MySQL:	SHOW TABLES;
SQL Server:	SELECT NAME FROM SYS.TABLES;
Oracle:	SELECT * FROM USER_TABLES;

3. 创建如下数据库用户：

```
Username: Steve
Password: Steve123
Access: learnsql database, SELECT on all tables
```

4. 根据你所使用的数据库实例，输入如下命令来列出数据库里的全部用户：

<b>MySQL:</b>	<b>SELECT * FROM USER;</b>
<b>SQL Server:</b>	<b>SELECT * FROM SYS.DATABASE_PRINCIPALS WHERE TYPE='S';</b>
<b>Oracle:</b>	<b>SELECT * FROM DBA_USERS;</b>

## [第七部分 摘要数据结构](#)

第20章 创建和使用视图及异名

第21章 使用系统目录

### [第20章 创建和使用视图及异名](#)

本章的重点包括：

什么是视图

如何使用视图

视图和安全

视图的存储

创建视图

结合视图

视图里的数据操作

什么是异名

管理异名

创建异名

删除异名

本章将介绍关于性能的一些知识，以及如何创建和删除视图，如何把视图用于安全管理，如何简化终端用户和报告的数据获取，最后还会讨论异名。

#### [20.1 什么是视图](#)

视图是一个虚拟表。也就是说，对于用户来说，视图在外观和行为上都类似表，但它不需要实际的物理存储。视图实际上是由预定义查询

形式的表所组成的。举例来说，从表EMPLOYEE\_TBL里创建一个视图，它只包含雇员的姓名和地址，而不是表里的全部字段。视图可以包含表的全部或部分记录，可以由一个表或多个表创建。

当创建一个视图时，实际上是在数据库里执行了一个SELECT语句，它定义了这个视图。这个SELECT语句可能只包含表里的字段名称，也可以包含各种函数和运算来操作或汇总给用户显示的数据。图20.1展示了视图的概念。

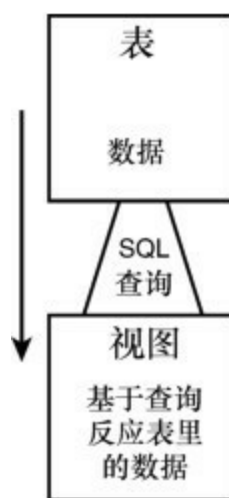


图20.1 视图

虽然视图并不占据实际的存储空间，但也被看作是一个数据库对象。视图与表之间的主要区别在于，表占据物理空间，而视图不需要物理空间，它只是从表里引用数据。

在数据库里，视图的使用方式与表是一样的，意味着我们可以像操作表一样从视图里获取数据。另外，我们还可以对视图里的数据进行操作，但存在着一定的限制。下面的小节将介绍视图的一些常见应用，以及视图在数据库里的保存方式。

**警告：删除用于创建视图的表**

如果用于创建视图的表被删除了，那么这个视图就不可访问了。如

果对这个视图做查询，就会收到错误信息。

### [20.1.1 使用视图来简化数据访问](#)

在有些情况下，通过数据库的归一化，或是数据库设计的过程，数据在表里的格式可能并不适合终端用户进行查询。这时，我们可以创建一系列的视图，让终端用户能够更简单地进行查询。举例来说，用户可能需要从数据库 `learnsql` 里查询雇员的薪水信息，但并不完全理解如何创建表 `EMPLOYEE_TBL` 和 `EMPLOYEE_PAY_TBL` 之间的结合。为了解决这个问题，我们可以创建一个视图来包含表的结合，让用户可以从这个视图获取数据。

### [20.1.2 使用视图作为一种安全角式](#)

提示：视图可以被用作一种安全角式

使用视图可以限制用户只访问表里的特定字段或满足一定条件的记录（在定义视图的 `WHERE` 子句里指定）。

视图可以作为数据库里的一种安全角式。假设我们有一个表 `EMPLOYEE_TBL`，它包含雇员姓名、地址、电话号码、紧急联系人、部门、职位、薪水或小时工资。在编写一些报告时，可能会有一些临时需求，要获取雇员的姓名、地址和电话号码。如果允许访问整个表，别人就可以看到每个雇员的收入是多少，这是我们所不允许的。为了防止这种情况发生，我们可以创建一个视图，让它只包含必要的信息：雇员姓名、地址和电话号码，并且让写报告的人可以使用这个视图，这样就可以避免他们访问表的其他敏感数据。

### [20.1.3 使用视图维护摘要数据](#)

假如摘要数据报告所基于的表经常更新，或是报告经常被创建，使用视图来包含摘要数据就是个很好的选择。

举例来说，有一个表包含个人信息，比如居住的城市、性别、薪水

和年龄。我们可以基于这个表来创建一个视图，统计每个城市的人员情况，比如平均年龄、平均薪水、男性总数、女性总数。在创建了视图之后，如果想获得这些信息，我们只需要对视图进行查询，而不需要使用复杂的SELECT语句。

利用摘要数据创建视图与从一个表或多个表创建视图的唯一区别就是使用了汇总函数。关于汇总函数的介绍请见第9章。

视图只保存在内存里，而且只需要保存其定义本身，这一点与其他数据库对象不同。视图由创建者或规划所有者所拥有。视图所有者自动拥有视图的全部权限，并且可以把视图的权限授予其他用户。对于视图来说，GRANT命令的GRANT OPTION权限的工作方式与表一样。关于权限的详细信息请见第19章。

## [20.2 创建视图](#)

视图是通过CREATE VIEW语句创建的。我们可以从一个表、多个表或另一个视图来创建视图。为了创建视图，用户必须拥有适当的系统权限。

基本的CREATE VIEW语法如下所示：

```
CREATE [RECURSIVE]VIEW VIEW_NAME
[COLUMN NAME [,COLUMN NAME]]
[OF UDT NAME [UNDER TABLE NAME]
[REF IS COLUMN NAME SYSTEM GENERATED |USER GENERATED | DERIVED]
[COLUMN NAME WITH OPTIONS SCOPE TABLE NAME]]
AS
{SELECT STATEMENT}
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

下面的几个小节分别介绍使用CREATE VIEW语句创建视图的不同方法。

注意：**ANSI SQL**不包含**ALTER VIEW**语句

大多数数据库实现里提供了ALTER VIEW语句，但ANSI SQL里没有。举例来说，在老版本的MySQL里，我们可以使用REPLACE VIEW语句修改当前视图。但是，最新版本的MySQL以及SQL Server和Oracle都支持ALTER VIEW语句。详细情况请参见具体实现的文档。

### 20.2.1 从一个表创建视图

我们可以从一个表创建视图。

语法如下所示：

```
CREATE VIEW VIEW_NAME AS
SELECT * | COLUMN1 [, COLUMN2 ]
FROM TABLE_NAME
[ WHERE EXPRESSION1 [, EXPRESSION2 ]]
[ WITH CHECK OPTION ]
[ GROUP BY ]
```

创建视图的最简单方式是基於单个表的全部内容，范例如下：

```
CREATE VIEW CUSTOMERS_VIEW AS
SELECT *
FROM CUSTOMER_TBL;
View created.
```

下面的范例从基表里只选择指定的字段，从而减少了视图里的内容：

```
CREATE VIEW EMP_VIEW AS
SELECT LAST_NAME, FIRST_NAME, MIDDLE_NAME
FROM EMPLOYEE_TBL;
View created.
```

下面的范例展示了如何利用基表里的多个字段组合成视图里的一个字段。利用SELECT子句里的别名，我们把视图字段命名为NAME。

```
CREATE VIEW NAMES AS
SELECT LAST_NAME || ', ' || FIRST_NAME || ' ' || MIDDLE_NAME NAME
FROM EMPLOYEE_TBL;
View created.
```

现在可以从视图NAMES里选择全部数据：

```
SELECT *
FROM NAMES;
NAME
-----
STEPHENS, TINA D
PLEW, LINDA C
GLASS, BRANDON S
GLASS, JACOB
WALLACE, MARIAH
SPURGEON, TIFFANY
6 rows selected.
```

下面的范例展示如何基于一个或多个表创建包含摘要数据的视图：

```
CREATE VIEW CITY_PAY AS
SELECT E.CITY, AVG(P PAY_RATE) AVG_PAY
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL P
WHERE E.EMP_ID = P.EMP_ID
GROUP BY E.CITY;
View created.
```

现在，从这个摘要视图里选择数据：

```
SELECT *
FROM CITY_PAY;
CITY          AVG_PAY
-----
GREENWOOD
INDIANAPOLIS   13.33333
WHITELAND
3 rows selected.
```



通过使用包含摘要数据的视图，针对基表的SELECT语句可以得到简化。

### [20.2.2 从多个表创建视图](#)

通过在SELECT语句里使用JOIN，我们可以从多个表创建视图。其语法如下：

```
CREATE VIEW VIEW_NAME AS
SELECT * | COLUMN1 [, COLUMN2 ]
FROM TABLE_NAME1, TABLE_NAME2 [, TABLE_NAME3 ]
WHERE TABLE_NAME1 = TABLE_NAME2
[ AND TABLE_NAME1 = TABLE_NAME3 ]
[ EXPRESSION1 ][, EXPRESSION2 ]
[ WITH CHECK OPTION ]
[ GROUP BY ]
```

下面是从多个表创建视图的范例：

```
CREATE VIEW EMPLOYEE_SUMMARY AS
SELECT E.EMP_ID, E.LAST_NAME, P.POSITION, P.DATE_HIRE, P.PAY_RATE
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL P
WHERE E.EMP_ID = P.EMP_ID;
View created.
```

在从多个表创建视图时，这些表必须在WHERE子句里通过共同字段实现结合。视图本身不过是一个SELECT语句而已，因此表在视图定义里的结合与在普通SELECT语句里是一样的。回忆一下，使用表的别名可以简化多表查询的可读性。

视图可以与表或其他视图相结合，其规则与表之间的结合一样。更多相关内容请见第13章。

### [20.2.3 从视图创建视图](#)

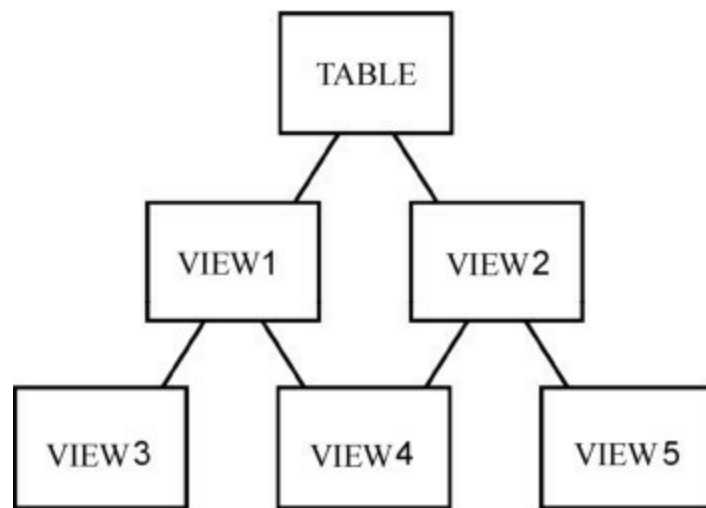
使用下面的语法可以从一个视图创建另一个视图：

```
CREATE VIEW2 AS  
SELECT * FROM VIEW1
```

视图创建视图可以具有多个层次（视图的视图的视图，以此类推），允许的层次取决于具体实现。基于视图创建视图的唯一问题在于它们的可管理性。举例来说，基于 VIEW1 创建了VIEW2，又基于VIEW2创建了VIEW3。如果VIEW1被删除了，VIEW2和VIEW3也就不可用了，因为支持这些视图的底层数据不存在了。因此，我们需要始终很好地理解数据库里的视图，以及它们依赖于什么数据库对象（参见图20.2）。

图20.2展示了视图基于表和其他视图的情况。VIEW1和VIEW2基于TABLE，VIEW3依赖于VIEW1，VIEW4依赖于VIEW1和VIEW2，VIEW5依赖于VIEW2。根据它们的关系，&nbsp;我们可以得出以下结论：

如果VIEW1被删除了，VIEW3和VIEW4就无效了；  
如果VIEW2被删除了，VIEW4和VIEW5就无效了；  
如果TABLE被删除了，这些视图就都无效了。



视图依赖

图20.2 视图依赖

注意：谨慎选择创建视图的方式

如果从基表和从另一个视图创建视图具有一样的难度和效率，那么我们首选从基表创建视图。

### **20.3 WITH CHECK OPTION**

这是CREATE VIEW语句里的一个选项，其目的是确保全部的UPDATE和INSERT语句满足视图定义里的条件。如果它们不满足条件，UPDATE或INSERT语句就会返回错误。WITH CHECK OPTION实际上通过查看视图定义是否被破坏来确保引用完整性。

下面是使用WITH CHECK OPTION创建视图的范例：

```
CREATE VIEW EMPLOYEE_PAGERS AS
SELECT LAST_NAME, FIRST_NAME, PAGER
FROM EMPLOYEE_TBL
WHERE PAGER IS NOT NULL
WITH CHECK OPTION;
View created.
```

在这个范例里，WITH CHECK OPTION会确保视图的PAGER字段里不包含NULL值，因为视图定义所依赖的数据里不允许在PAGER字段里包含NULL值。

尝试在PAGER字段里插入一个NULL值：

```
INSERT INTO EMPLOYEE_PAGERS
VALUES ('SMITH', 'JOHN', NULL);

insert into employee_pagers
*
ERROR at line 1:
ORA-01400: mandatory (NOT NULL) column is missing or NULL during insert
```

在基于视图创建另一个视图时，WITH CHECK OPTION有两个选项：CASCADDED和LOCAL，其中CASCADDED是默认选项。CASCADDED是ANSI标准语法。但是，Microsoft SQL Server和Oracle使用稍有不同的CASCADE关键字。在对基表进行更新时，CASCADDED选项会检查所有底层视图、所有完整性约束，以及新视图的定义条件。LOCAL 选项只检查两个视图的完整性约束和新视图的定义条件，不检查底层的表。因此，使用CASCADDED选项创建视图是更安全的作法，基表的引用完整性也得到了保护。

## [20.4 从视图创建表](#)

我们可以从视图创建一个表，就像从一个表创建另一个表（或从一个视图创建另一个视图）一样。

语法如下：

```
CREATE TABLE TABLE_NAME AS
SELECT { * | COLUMN1 [, COLUMN2 ]
FROM VIEW_NAME
[ WHERE CONDITION1 [, CONDITION2 ]
[ ORDER BY ]
```

注意：表与视图的细微差别

表与视图的主要区别在于表包含实际的数据、占据物理存储空间，而视图不包含数据，而且只需要保存视图定义（查询语句）。

首先，基于两个表创建一个视图：

```
CREATE VIEW ACTIVE_CUSTOMERS AS
SELECT C.*
FROM CUSTOMER_TBL C,
      ORDERS_TBL O
WHERE C.CUST_ID = O.CUST_ID;
View created.
```

接下来，基于前面这个视图创建一个表：

```
CREATE TABLE CUSTOMER_ROSTER_TBL AS
SELECT CUST_ID, CUST_NAME
FROM ACTIVE_CUSTOMERS;
Table created.
```

注意：在查询视图时使用**ORDER BY**子句

与在 CREATE VIEW 语句里使用 GROUP BY 子句相比，在查询视图的SELECT语句里使用ORDER BY子句更简单、效果更好。

最后，从这个表里选择数据：

```
SELECT *
FROM CUSTOMER_ROSTER_TBL;
CUST_ID    CUST_NAME
.....
232        LESLIE GLEASON
12         MARYS GIFT SHOP
43         SCHYLERS NOVELTIES

090        WENDY WOLF
287        GAVINS PLACE
432        SCOTTYS MARKET

6 rows selected.
```

## [20.5 视图与ORDER BY子句](#)

CREATE VIEW语句里不能包含ORDER BY子句，但是GROUP BY子句用于CREATE VIEW语句时，可以起到类似ORDER BY子句的作用。

下面是在CREATE VIEW语句里使用GROUP BY子句的范例：

```
CREATE VIEW NAMES2 AS
SELECT LAST_NAME || ', ' || FIRST_NAME || ' ' || MIDDLE_NAME NAME
FROM EMPLOYEE_TBL
GROUP BY LAST_NAME || ', ' || FIRST_NAME || ' ' || MIDDLE_NAME;
View created.
```

如果从这个视图里选择全部数据，它们是以字母顺序排列的（因为根据NAME进行了分组）。

```
SELECT *
FROM NAMES2;
NAME
-----
GLASS, BRANDON S
GLASS, JACOB
PLEW, LINDA C
SPURGEON, TIFFANY
STEPHENS, TINA D
WALLACE, MARIAH

6 rows selected.
```

## [20.6 通过视图更新数据](#)

在一定条件下，视图的底层数据可以进行更新：

视图不包含结合；

视图不包含GROUP BY子句；

视图不包含UNION语句；

视图不包含对伪字段ROWNUM的任何引用；

视图不包含任何组函数；

不能使用DISTINCT子句；

WHERE子句包含的嵌套的表表达式不能与FROM子句引用同一个表。

视图可以执行INSERT、UPDATE和DELETE等语句，

关于UPDATE命令的语法请见第14章。

## [20.7 删除视图](#)

DROP VIEW命令用于从数据库里删除视图，它有两个选项：RESTRICT和CASCADE。如果使用了RESTRICT选项进行删除操作，而其他视图在约束里有所引用，删除操作就会出错。如果使用了CASCADE选项，而且其他视图或约束被引用了，DROP VIEW也会成功，&nbsp;而且底层的视图或约束也会被删除。范例如下：

```
DROP VIEW NAMES2;  
View dropped.
```

## [20.8 嵌套视图对性能的影响](#)

在查询中使用视图，与使用表有着相同的性能特性。因此，用户必须意识到，在视图中隐藏复杂的逻辑会导致系统需要查询底层表来分析并组合数据。在进行性能调整的时候，视图需要和其他SQL语句一样被调整。如果构成视图的查询没有经过事先设计，那么视图本身就会对性能产生影响。

此外，有些用户将查询分解为各种视图构成的多个单元，以便简化复杂的查询。这种方法在简化复杂逻辑方面看起来是个好办法，但却会降低性能。因为搜索发动机需要分析每一层的视图，来确定为了完成搜索需要进行哪些工作。

嵌套的层数越多，搜索发动机为了获得一个执行计划而需要进行的分析工作就越多。实际上，大多数搜索发动机无法确保获得一个完美的执行计划，而只能保证执行一个耗时最短的计划。因此，最好的方法就是，尽量减少代码中的嵌套层数，并且测试并调整创建视图所用到的语句。



注意：异名不属于**ANSI SQL**标准

异名并不属于ANSI SQL标准，但由于多个主流实现都在使用它，我们最好还是在此讨论一下。关于异名的使用请查看具体实现的文档。MySQL不支持异名，但我们可以使用视图来实现同样的功能。

## [20.9 什么是异名](#)

异名就是表或视图的另一个名称。我们创建别名通常是为了在访问其他用户的表或视图时不必使用完整限制名。异名可以创建为PUBLIC或PRIVATE，PUBLIC的异名可以被数据库里的其他用户使用，而PRIVATE异名只能被所有者和拥有权限的用户使用。

异名由数据库管理员（或某个指定的人员）或个人用户管理。由于异名有两种类型：PUBLIC和PRIVATE，在创建异名时可能需要不同的系统级权限。一般来说，全部用户都可以创建PRIVATE异名，而只有数据库管理员（DBA）或被授权的用户可以创建PUBLIC异名，详细情况请参见具体实现的文档。

### [20.9.1 创建异名](#)

创建异名的一般语法如下所示：

```
CREATE [PUBLIC|PRIVATE] SYNONYM SYNONYM_NAME FOR TABLE|VIEW
```

下面的范例为表CUSTOMER\_TBL创建一个异名：CUST，之后我们再引用这个表就不用输入完整的表名了。



```
CREATE SYNONYM CUST FOR CUSTOMER_TBL;
```

```
Synonym created.
```

```
SELECT CUST_NAME
```

```
FROM CUST;
```

```
CUST_NAME
```

```
-----
```

```
LESLIE GLEASON
```

```
NANCY BUNKER
```

```
ANGELA DOBKO
```

```
WENDY WOLF
```

```
MARYS GIFT SHOP
```

```
SCOTTYS MARKET
```

```
JASONS AND DALLAS GOODIES
```

```
MORGANS CANDIES AND TREATS
```

```
SCHYLERS NOVELTIES
```

```
GAVINS PLACE
```

```
HOLLYS GAMEARAMA
```

```
HEATHERS FEATHERS AND THINGS
```

```
RAGANS HOBBIES INC
```

```
ANDYS CANDIES
```

```
RYANS STUFF
```

```
15 rows selected.
```

异名的另一个常见应用是，表的所有者给表创建一个异名，这样其他有权限访问这个表的用户不必在表的名称前面添加所有者名称也可以引用这个表了。

```
CREATE SYNONYM PRODUCTS_TBL FOR USER1.PRODUCTS_TBL;
```

```
Synonym created.
```

### [20.9.2 删除异名](#)

删除异名与删除其他数据库对象很类似，一般语法如下所示：

```
DROP [PUBLIC|PRIVATE] SYNONYM SYNONYM_NAME
```

范例如下：

```
DROP SYNONYM CUST;  
Synonym dropped.
```

## 20.10 小结

本章讨论了SQL里两个重要特性：视图和异名。在很多情况下，这些能够对关系型数据库用户有所帮助的特性并没有得到充分应用。视图就是一个虚拟表——外观与行为都像表，但不像表那样占据物理空间。视图实际上是由对表和其他视图的查询所定义的，其主要用于限制用户能够查看的数据、简化数据和进行数据摘要。视图可以基于视图创建，但要注意不要嵌套太多的层次，以避免失去对它们的管理控制。创建视图时有多个选项，有些实现还具有自己特殊的选项。

异名也是数据库里的对象，它代表着其他对象。我们可以创建一个短的异名来代表名称较长的对象，或是代表被其他用户所拥有的对象，从而简化数据库里对象名称的使用。异名有两种类型：**PUBLIC** 和 **PRIVATE**。**PUBLIC** 异名可以被数据库里的全部用户访问，而 **PRIVATE** 异名只被单个用户访问。**DBA** 通常负责创建 **PUBLIC** 异名，而个人用户通常创建自己的 **PRIVATE** 异名。

## 20.11 问与答

问：视图怎么能包含数据而又不占据存储空间呢？

答：视图不包含数据，它是一个虚拟表，或是一个存储的查询。视图所需的空间只是定义语句所需要的。

问：如果视图所基于的视图被删除了，它会怎么样？

答：这个视图就无效了，因为底层的数据已经不存在了。

问：在创建异名时，其名称有什么限制？

答：这取决于具体的实现。在大多数主流实现里，异名的命名规则

与数据库里表和其他对象的命名规则一样。

## **20.12 实践**

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### **20.12.1 测验**

1. 在一个基于多个表创建的视图里，我们可以删除记录吗？
2. 在创建一个表时，所有者会自动被授予适当的权限。在创建视图时也是这样吗？
3. 在创建视图时，使用什么子句对数据进行排序？
4. 在基于视图创建视图时，使用什么选项检查完整性约束？
5. 在尝试删除视图时，由于存在着多个底层视图，操作出现了错误。这时怎样做才能删除视图？

### **20.12.2 练习**

1. 编写一个语句，基于表EMPLOYEE\_TBL的全部内容创建一个视图。
2. 编写一个语句创建一个包含摘要数据的视图，显示表EMPLOYEE\_TBL 里每个城市的平均小时工资和平均薪水。
3. 再次创建练习2中的摘要数据视图，但不要使用表EMPLOYEE\_TBL，而是使用练习1中所创建的视图。比较两个结果。
4. 使用练习2中创建的视图来创建一个名为EMPLOYEE\_PAY\_SUMMARIZED的表。想办法确定视图和表拥有相同的数据。

5. 编写SQL语句来删除表和刚刚创建的视图。

## [第21章 使用系统目录](#)

本章的重点包括：

什么是系统目录

如何创建系统目录

系统目录里包含什么数据

系统目录表的范例

查询系统目录

更新系统目录

本章介绍系统目录，在某些关系型数据库的实现里，这也被称为数据目录。本章将介绍系统目录的作用与内容，以及如何对它进行查询来获得数据库的信息。每种主流实现都具有某种形式的系统目录，保存了关于数据库本身的信息。本章中将展示书中所涉及的不同实现里系统目录所包含的元素。

### [21.1 什么是系统目录](#)

系统目录是一些表和视图的集合，它们包含了关于数据库的信息。每个数据库都有系统目录，其中定义了数据库的结构，还有数据库所包含数据的信息。举例来说，用于数据库里所有表的数据目录语言（DDL）就保存在系统目录里。图21.1展示了数据库的系统目录。

从图21.1可以看出，系统目录实际上是数据库的组成部分。数据库里包含的是对象，比如表、索引和视图。系统目录基本上就是一组对象，包含了定义数据库里其他对象的信息、数据库本身的结构以及其他各种重要信息。

在具体实现里，系统目录的内容会被划分为对象的逻辑组，以表的形式供数据库管理员（DBA）和其他数据库用户访问。举例来说，某个用户可能需要查看自己具有的数据库权限，但不需要知道数据库内部结构或进程。普通用户通常可以对系统目录进行查询来获得关于所拥有对象和权限的信息，而 DBA 需要能够获取数据库里任何结构或事件的信息。在某些实现里，有些系统目录的对象只能由DBA访问。

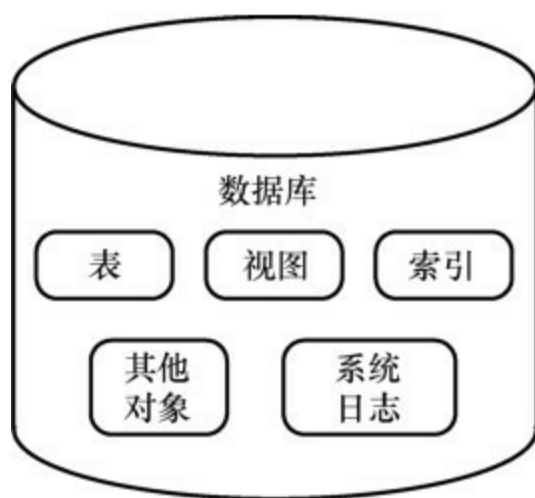


图21.1 系统目录

系统目录对于 DBA 或其他需要了解数据库结构和特征的用户来说都是非常重要的。在数据库用户没有使用GUI界面时，它尤为重要。系统目录不仅允许DBA和用户，而且允许数据库服务程序本身对其进行维护。

注意：不同实现的系统目录有所不同

每个实现对系统目录的表和视图都具有自己的命名规则。名称本身并不重要，重要的是功能，以及它包含什么内容、如何检索这些内容。

## [21.2 如何创建系统目录](#)

系统目录或者是在数据库创建时自动创建的，或是由 DBA 在数据

库创建之后立即创建的。举例来说，Oracle数据库会运行一系列由厂商提供的预定义SQL脚本，创建系统目录里全部的表格和视图。系统目录的表格和视图是由系统所拥有的，不属于任何规划。举例来说，在Oracle里，系统目录的所有者是一个名为SYS的用户，它对数据库具有完全的权限。在Microsoft SQL Server里，SQL服务程序的系统目录位于master数据库里。在MySQL里，系统目录位于mysql数据库里。系统目录保存的实际位置请查看具体实现的文档。

### [21.3 系统目录里包含什么内容](#)

系统目录里包含多种信息，可以被很多用户访问，但有时不同用户的使用目的并不一样。

系统目录包含的内容有：

用户账户和默认设置；

权限和其他安全信息；

性能统计；

对象大小估计；

对象变化；

表结构和存储；

索引结构和存储；

数据库其他对象的信息，比如视图、异名、触发器和存储过程；

表约束和引用完整性信息；

用户会话；

审计信息；

内部数据库设置；

数据库文件的位置。

系统目录由数据库服务程序维护。举例来说，当一个表被创建时，

数据库服务程序把数据插入到系统目录里适当的表或视图。当表的结构被修改时，系统目录里相应的对象也被更新。下面这些小节分别介绍系统目录里所包含的信息。

### [21.3.1 用户数据](#)

关于个人用户的全部信息都保存在系统目录里：用户具有的系统和对象权限、用户拥有的对象、用户不拥有但能够访问的对象。用户可以通过查询访问用户表或视图。关于系统目录对象的详细情况请参见具体实现的文档。

### [21.3.2 安全信息](#)

系统目录也保存安全信息，比如用户标识、加密的密码、各种权限和权限组。有些实现里包含审计表，可以跟踪数据库发生的事件，包括由谁引发、何时等信息。在很多实现里，利用系统目录还可以密切监视数据库用户会话。

### [21.3.3 数据库设计信息](#)

系统目录包含关于数据库的信息，包括数据库的创建日期、名称、对象大小估计、数据文件的大小和位置、引用完整性信息、索引、每个表的字段信息和属性。

### [21.3.4 性能统计](#)

性能统计一般也在系统目录里，包括关于 SQL 语句性能的信息，比如优化器执行 SQL 语句的时间和方式。其他性能信息还有内存分配和使用、数据库里剩余空间、控制表格和索引碎片的信息。利用这些信息可以调整数据库，重新安排 SQL 查询，重新设计访问数据的方法，从而得到更好的整体性能和更快的 SQL 查询响应。

## [21.4 不同实现里的系统目录表格](#)

每个实现都有一些表格和视图来构成系统目录，有些还分为用户级、系统级和DBA级。

关于系统目录表格的详细情况请参见具体实现的文档，表21.1列出了主流实现的范例。

表21.1 主流实现的系统目录对象



Microsoft SQL Server	
表格名称	内容
SYSUSERS	数据库用户
SYS.DATABASES	全部数据库片断
SYS.DATABASE_PERMISSIONS	全部数据库权限
SYS.DATABASE_FILES	全部数据库文件
SYSINDEXES	全部索引
SYSCONSTRAINTS	全部约束
SYS.TABLES	全部数据库表
SYS.VIEWS	全部数据库视图
Oracle	
表格名称	内容
ALL_TABLES	用户访问的表
USER_TABLES	用户拥有的表
DBA_TABLES	数据库里全部表
DBA_SEGMENTS	片断存储
DBA_INDEXES	全部索引
DBA_USERS	数据库里的全部用户
DBA_ROLE_PRIVS	分配的角色
DBA_ROLES	数据库里的角色
DBA_SYS_PRIVS	分配的系统权限
DBA_FREE_SPACE	数据库剩余空间
V\$DATABASE	数据库的创建
V\$SESSION	当前会话
MySQL	
表格名称	内容
COLUMNS_PRIV	字段权限
DB	数据库权限
FUNC	自定义函数的管理
HOST	与 MySQL 相关联的主机名称
TABLES_PRIV	表权限
USER	表关系

上面只是列出了书中涉及的一些关系型数据库实现里的部分系统目录对象，主要是比较类似的对象。每个实现在系统目录内容的组织方面都是很独特的。

## 21.5 查询系统目录

我们可以像对待数据库里的其他表格和视图一样使用 SQL 查询系统目录里的表格和视图。用户通常只能查询与用户相关的表，不能访问系统表，后者通常只能由被授权的用户访问，比如DBA。

创建查询从系统目录里获取数据与对数据库的其他表格进行操作是一样的。举例来说，下面的查询从Microsoft SQL Server的表SYS.TABLES里返回全部记录：

```
SELECT * FROM SYS.TABLES;  
GO
```

下面的查询返回数据库里的全部用户账户，它运行于MySQL系统数据库上：

```
SELECT USER  
FROM ALL_USER;  
USER  
-----  
ROOT  
SYSTEM  
RYAN  
SCOTT  
DEMO  
RON  
USER1  
USER2  
8 rows selected.
```

注意：有关下面的范例

下面的范例使用MySQL的系统目录。在此选择使用MySQL没有特别的意图，只是选择了本书所涉的一种数据库实现而已。

下面的范例列出我们的learnsql规划里的全部表格，它运行于

information\_schema上:

```
SELECT TABLE_NAME
FROM TABLES WHERE TABLE_SCHEMA='learnsql';
TABLE_NAME
-----
CUSTOMER_TBL
EMPLOYEE_PAY_TBL
EMPLOYEE_TBL
PRODUCTS_TBL
ORDERS_TBL
5 rows selected.
```

警告：不要手动修改系统目录中的表

不要以任何方式直接操作系统目录里的表（只有DBA能够操作系统目录的表），否则可能会破坏数据库的完整性。与数据库结构有关的信息，以及数据库的全部对象都保存在系统目录中，它通常是与数据库里的其他数据隔离的。有些实现，例如Microsoft SQL Server，不允许用户手动修改系统目录中的表，以确保系统的完整性。

下面的查询返回数据库用户BRANDON的全部系统权限：

```
SELECT GRANTEE, PRIVILEGE_TYPE
FROM USER_PRIVILEGES
WHERE GRANTEE = 'BRANDON';
```

GRANTEE	PRIVILEGE
BRANDON	SELECT
BRANDON	INSERT
BRANDON	UPDATE
BRANDON	CREATE

4 rows selected.

注意：本小节所涉信息有限

本小节范例中返回的信息与实际系统目录相比简直是九牛一毛。在

实际工作中，最好把系统目录返回的信息输出到文件，打印出来作为参考。关于系统目录里表格以及表格内字段的详细情况请参见具体实现的文档。

## 21.6 更新系统目录对象

系统目录只能执行查询操作——DBA也是如此。系统目录的更新是由数据库服务程序自动完成的。举例来说，当用户发出CREATE TABLE命令时，数据库里会创建一个表，数据库服务程序就会把创建这个表的DDL放到系统目录里适当的表里。

系统目录里的表从不需要进行手工更新，即使用户有这个能力也不需要。数据库服务程序会根据数据库里发生的行为对系统目录进行相应的更新，如图21.2所示。

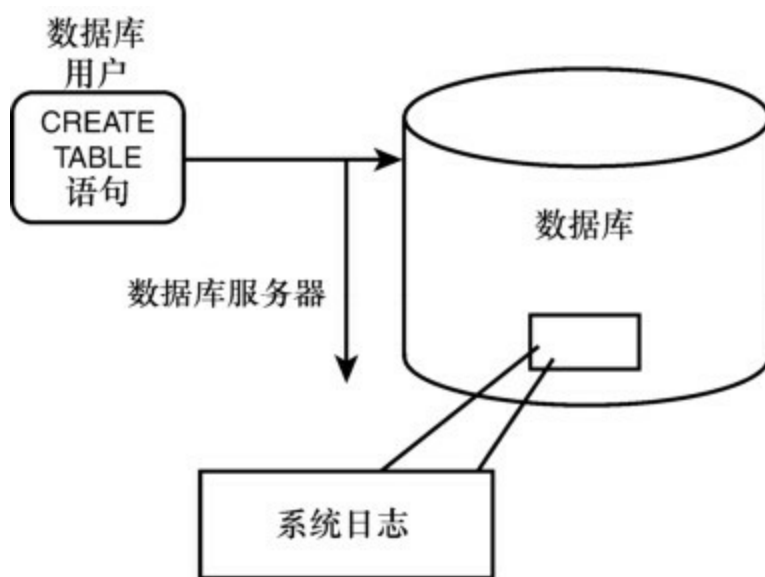


图21.2 更新系统目录

## 21.7 小结

本章介绍了关系型数据库的系统目录。从某种意义上来说，系统目录

是数据库里的数据库，包含了与其所在数据库相关的全部信息，用于维护数据库的整体结构、跟踪数据库里发生的事件与改变、为数据库整体管理提供各种信息。系统目录只能执行查询操作，没有用户应该对系统目录进行直接修改。然而，在数据库结构本身发生任何一次变化时，比如创建表，数据库服务程序都会自动对系统目录进行更新。

## [21.8 问与答](#)

问：作为一名数据库用户，我知道可以获取我的对象的信息，但如何才能得到其他用户的对象的信息呢？

答：用户可以利用一组表或视图对系统目录的大部分信息进行查询，其中就包括所访问对象的信息。为了了解其他用户的情况，我们需要查看包含相应内容的系统目录。举例来说，在Oracle里，我们可以查看系统目录DBA\_TABLES和DBA\_USERS。

问：如果用户忘记了密码，**DBA**是否可以通过查询某个表而获得这个密码呢？

答：是，也不是。密码被保存在一个系统表格里，通常是加密的，所以即使 **DBA** 也不能读取这个密码。如果用户忘记了密码，密码就必须被重置，而这是**DBA**能够轻易完成的。

问：如何了解系统目录表格里包含了什么字段？

答：系统目录表格可以像其他表格一样被查询，所以只要查询适当的表就可以获得这个信息。

## [21.9 实践](#)

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答

案请见附录C。

### 21.9.1 测验

1. 在某些实现里，系统目录也被称为什么？
2. 普通用户能够更新系统目录吗？
3. 在Microsoft SQL Server里，哪个系统表格包含了数据库里视图的信息？
4. 谁拥有系统目录？
5. Oracle数据对象ALL\_TABLES和DBA\_TABLES之间的区别是什么？
6. 谁修改系统表格？

### 21.9.2 练习

1. 在第19章中，我们查看了mysql里的系统表格。现在查看一些本章中讨论过的系统表，复习一下这些表格。
2. 在提示符下，输入命令来获取以下信息：  
所有表的信息。  
所有视图的信息。  
数据库中所有的用户名。
3. 使用多个系统表来编写查询，获得learnsql数据库中的所有用户及其权限。

## [第八部分 在实际工作中应用SQL知识](#)

第22章 高级SQL主题

第23章 SQL扩展到企业、互联网和内部网

第24章 标准SQL的扩展

### [第22章 高级SQL主题](#)

本章的重点包括：

什么是光标

使用存储过程

什么是触发器

动态SQL基础

使用SQL生成SQL

直接SQL与嵌入SQL

调用级接口

前面的章节介绍了SQL的一些基本操作，比如从数据库查询数据、创建数据库结构、操作数据库里的数据，现在我们来介绍一些高级SQL主题，内容包括光标、存储过程、触发器、动态SQL、直接SQL与嵌入SQL、SQL生成SQL。很多SQL实现都支持这些高级特性，增强了SQL的功能。

注意：某些主题不属于**ANSI SQL**

某些主题并不都属于ANSI SQL，所以其实际语句和规则要取决于具体实现。本章会介绍一些主要厂商的语法以供比较。

#### [22.1 光标](#)

通常，数据库操作被认为是以数据集为基础的操作。这就意味着，大部分ANSI SQL命令是作用于一组数据的。但是，光标则被用于通过以记录为单位的操作，来获得数据库中数据的子集。因此，程序可以依次对光标里的每一行进行求值。光标一般用于过程化程序里嵌入的SQL语句。有些光标是由数据库服务程序自动隐含创建的，有些是由SQL程序员定义的。每个SQL实现里对光标用法的定义是不同的。

下面介绍本书中一直在应用的 3个流行SQL实现的范例：MySQL、SQL Server和Oracle。

MySQL里对光标的声明语法如下所示：

```
DECLARE CURSOR_NAME CURSOR  
FOR SELECT_STATEMENT
```

SQL Server里对光标的声明语法如下所示：

```
DECLARE CURSOR_NAME CURSOR  
FOR SELECT_STATEMENT  
[ FOR [READ ONLY | UPDATE {[ COLUMN_LIST ]}]
```

Oracle的语法如下：

```
DECLARE CURSOR CURSOR_NAME  
IS {SELECT_STATEMENT}
```

下面的光标包含了表EMPLOYEE\_TBL全部记录的子集：

```
DECLARE CURSOR EMP_CURSOR IS  
SELECT * FROM EMPLOYEE_TBL  
{ OTHER PROGRAM STATEMENTS }
```

根据ANSI标准，在光标被创建之后，可以使用如下操作对其进行访问。



OPEN: 打开定义的光标。

FETCH: 从光标获取记录，赋予程序变量。

CLOSE: 在对光标的操作完成之后，关闭光标。

### [22.1.1 打开光标](#)

要使用光标，必须首先打开光标。当光标被打开时，指定光标的SELECT语句被执行，查询的结果被保存在内存里的特定区域。

在MySQL和Microsoft SQL Server中打开一个光标的语法如下：

```
OPEN CURSOR_NAME
```

在Oracle里的语法如下：

```
OPEN CURSOR_NAME [ PARAMETER1 [, PARAMETER2 ]]
```

下面的范例会打开光标EMP\_CURSOR：

```
OPEN EMP_CURSOR
```

### [22.1.2 从光标获取数据](#)

在光标打开之后，我们可以使用FETCH语句获取光标的内容（查询的结果）。

在SQL Server里，FETCH语句的语法如下所示：

```
FETCH NEXT FROM CURSOR_NAME [ INTO FETCH_LIST ]
```

在Oracle里的语法如下：

```
FETCH CURSOR_NAME {INTO : HOST_VARIABLE  
[[ INDICATOR ] : INDICATOR_VARIABLE ]  
[, : HOST_VARIABLE  
[[ INDICATOR ] : INDICATOR_VARIABLE ]]  
| USING DESCRIPTOR DESCRIPTOR }
```

MySQL里的语法如下：

```
FETCH CURSOR_NAME into VARIABLE_NAME,[VARIABLE_NAME] ...
```

下面的FETCH语句把光标EMP\_CURSOR里的内容获取到变量EMP\_RECORD:

```
FETCH EMP_CURSOR INTO EMP_RECORD
```

在从光标获得数据时，需要注意可能会到达光标末尾。不同的实现使用不同的方法来解决这个问题，从而避免用户在关闭光标的时候产生错误。下面是一些伪代码实例，显示了MySQL、Microsoft SQL Server和Oracle如何处理这种情况，帮助读者理解光标的处理过程。

MySQL中的语法如下：

```
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE custname VARCHAR(30);
    DECLARE namecursor CURSOR FOR SELECT CUST_NAME FROM TBL_CUSTOMER;

    OPEN namecursor;
    read_loop: LOOP
        FETCH namecursor INTO custname;
        IF done THEN
            LEAVE read_loop;
        END IF;
        -- Do something with the variable
    END LOOP;
    CLOSE namecursor;
END;
```

Microsoft SQL Server中的语法如下：

```

BEGIN
    DECLARE @custname VARCHAR(30);
    DECLARE namecursor CURSOR FOR SELECT CUST_NAME FROM TBL_CUSTOMER;
    OPEN namecursor;
    FETCH NEXT FROM namecursor INTO @custname
    WHILE (@@FETCH_STATUS<>-1)
        BEGIN
            IF (@@FETCH_STATUS<>-2)
                BEGIN
                    -- Do something with the variable
                END
            FETCH NEXT FROM namecursor INTO @custname
        END
    CLOSE namecursor
    DEALLOCATE namecursor
END;

```

Oracle中的语法如下：

```

custname varchar(30);
CURSOR namecursor
IS
SELECT CUST_NAME FROM TBL_CUSTOMER;
BEGIN
    OPEN namecursor;
    FETCH namecursor INTO custname;
    IF namecursor%notfound THEN
        -- Do some handling as you are at the end of the cursor
    END IF;
    -- Do something with the variable
    CLOSE namecursor;
END;

```

### [22.1.3 关闭光标](#)

光标可以打开，当然就可以关闭。在光标关闭之后，程序就不能再使用它了。关闭光标是相当简单的。

下面是SQL Server里关闭和释放光标的语法：

```
CLOSE CURSOR_NAME  
DEALLOCATE CURSOR CURSOR_NAME
```

在Oracle里，当光标被关闭之后，不必使用DEALLOCATE语句就可以释放资源和姓名。其语法如下：

```
CLOSE CURSOR_NAME
```

MySQL的光标也是这样，不必使用DEALLOCATE语句。其语法如下：

```
CLOSE CURSOR_NAME
```

注意：高级特性在不同实现间的差别很大

从前面的范例可以看出，不同实现之间的差别很大，特别是高级特性和 SQL 扩展（详情请见第24章）。关于光标使用的详细情况请参见具体实现的文档。

## [22.2 存储过程和函数](#)

注意：释放光标所占据的资源

关闭光标并不一定意味着会释放它所占据的内存空间。在某些实现里，光标占用的内存必须使用DEALLOCATE语句才能解除分配。当光标被解除分配时，相关联的内存被释放，而光标的名称可以被再次使用。而在某些实现里，当光标被关闭时，内存会被隐含地解除分配。当光标占据的内容被释放之后，它们可以用于其他操作，比如打开另一个光标。

存储过程是一组相关联的SQL语句，通常被称为函数和子程序，能够让程序员更轻松和灵活地编程。这是因为存储过程与一系列单个SQL

语句相比更容易执行。存储过程可以嵌套在另一个存储过程里，也就是说存储过程可以调用其他存储过程，后者又可以调用另外的存储过程，依此类推。

利用存储过程可以实现过程化编程。基本的SQL DDL（数据定义语言）、DML（数据管理语言）和 DQL（数据查询语言）语句

（CREATE TABLE、INSERT、UPDATE、SELECT等）只是告诉数据库需要做什么，而不是如何去做。而通过对存储过程进行编程，我们就可以告诉数据库发动机如何处理数据。

存储过程是保存在数据库里的一组SQL语句或函数，它们被编译，随时可以被数据库用户使用。存储函数与存储过程是一样的，但函数可以返回一个值。

函数由过程调用。当函数被过程调用时也可以传递参数，函数会进行所需要的计算，并且把一个值返回给调用它的过程。

当存储过程被创建之后，组成它的各种子程序和函数都保存在数据库里。这些存储过程经过了预编译，可以随时由用户调用。

下面是MySQL创建存储过程的语法：

```
CREATE [ OR REPLACE ] PROCEDURE PROCEDURE_NAME
[ (ARGUMENT [{IN | OUT | IN OUT} ] TYPE,
  ARGUMENT [{IN | OUT | IN OUT} ] TYPE) ] { AS }
PROCEDURE_BODY
```

下面是SQL Server创建存储过程的语法：

```

CREATE PROCEDURE PROCEDURE_NAME
[ [ ( ) @PARAMETER_NAME
DATATYPE [ (LENGTH) | (PRECISION) [, SCALE ] )
[ = DEFAULT ] [ OUTPUT ] ]
[, @PARAMETER_NAME
DATATYPE [ (LENGTH) | (PRECISION) [, SCALE ] )
[ = DEFAULT ] [ OUTPUT ] ] [ ) ] ]
[ WITH RECOMPILE ]
AS SQL_STATEMENTS

```

Oracle的语法如下所示：

```

CREATE [ OR REPLACE ] PROCEDURE PROCEDURE_NAME
[ ( ARGUMENT [{ IN | OUT | IN OUT } ] TYPE,
ARGUMENT [{ IN | OUT | IN OUT } ] TYPE) ] { IS | AS }
PROCEDURE_BODY

```

下面是一个很简单的存储过程，它在表PRODUCTS\_TBL里插入一行新记录：

```

CREATE PROCEDURE NEW_PRODUCT
( PROD_ID IN VARCHAR2, PROD_DESC IN VARCHAR2, COST IN NUMBER )
AS
BEGIN
    INSERT INTO PRODUCTS_TBL
    VALUES ( PROD_ID, PROD_DESC, COST );
    COMMIT;
END;
Procedure created.

```

SQL Server里执行存储过程的语法如下：

```

EXECUTE [ @RETURN_STATUS = ]
PROCEDURE_NAME
[ [ @PARAMETER_NAME = ] VALUE |
[ @PARAMETER_NAME = ] @VARIABLE [ OUTPUT ] ]
[ WITH RECOMPILE ]

```

下面是Oracle的语法：

```
EXECUTE [ @RETURN STATUS = ] PROCEDURE NAME  
[[ @PARAMETER NAME = ] VALUE | [ @PARAMETER NAME = ] @VARIABLE [ OUTPUT ]]  
[ WITH RECOMPILE ]
```

下面是MySQL的语法：

```
CALL PROCEDURE_NAME([PARAMETER[,.....]])
```

注意：基本**SQL**命令往往是相同的

可以看出，不同 SQL 实现里对过程进行编程的语法有很大的差别。在不同的SQL实现里，基本的SQL命令应该是相同的，但编程概念（变量、条件语句、光标、循环）可能会有很大不同。

现在执行前面创建的过程：

```
CALL NEW_PRODUCT ('9999','INDIAN CORN',1.99);  
PL/SQL procedure successfully completed.
```

与单个SQL语句相比，存储过程具有一些明显的优点，包括：

存储过程的语句已经保存在数据库里了；

存储过程的语句已经被解析过，以可执行格式存在；

存储过程支持模块化编程；

存储过程可以调用其他存储过程和函数；

存储过程可以被其他类型的程序调用；

存储过程通常具有更好的响应时间；

存储过程提高了整体易用性。

## [22.3 触发器](#)

触发器是数据库里编译了的SQL过程，基于数据库里发生的其他行为来执行操作。它是存储过程的一种，会在特定 DML 行为作用于表格时被执行。它可以在 INSERT、DELECT或UPDATE语句之前或之后执

行，可以在这些语句之前检查数据完整性，可以回退事务，可以修改一个表里的数据，可以从另一个数据库的表里读取数据。

在大多数情况下，触发器都是很不错的函数，但它们会导致更多的I/O开销。如果使用存储过程或程序能够在较少开销下完成同样的工作，就应该尽量不使用触发器。

### 22.3.1 CREATE TRIGGER语句

这个语句用于创建触发器。

ANSI标准语法是：

```
CREATE TRIGGER TRIGGER NAME
[[BEFORE | AFTER] TRIGGER EVENT ON TABLE NAME]
[REFERENCING VALUES ALIAS LIST]
[TRIGGERED ACTION
TRIGGER EVENT ::=
INSERT | UPDATE | DELETE [OF TRIGGER COLUMN LIST]
TRIGGER COLUMN LIST ::= COLUMN NAME [, COLUMN NAME]
VALUES ALIAS LIST ::=
VALUES ALIAS LIST ::=
OLD [ROW] ' OLD VALUES CORRELATION NAME |
NEW [ROW] ' NEW VALUES CORRELATION NAME |
OLD TABLE ' OLD VALUES TABLE ALIAS |
NEW TABLE ' NEW VALUES TABLE ALIAS
OLD VALUES TABLE ALIAS ::= IDENTIFIER
NEW VALUES TABLE ALIAS ::= IDENTIFIER
TRIGGERED ACTION ::=
[FOR EACH [ROW | STATEMENT] [WHEN SEARCH CONDITION]]
TRIGGERED SQL STATEMENT
TRIGGERED SQL STATEMENT ::=
SQL STATEMENT | BEGIN ATOMIC [SQL STATEMENT;]
END
```

MySQL里使用触发器的语法是：



```
CREATE [DEFINER={user | CURRENT_USER }]
TRIGGER TRIGGER_NAME
{BEFORE | AFTER }
{ INSERT | UPDATE | DELETE [, ..] }
ON TABLE_NAME
AS
SQL_STATEMENTS
```

SQL Server里创建触发器的语法是：

```
CREATE TRIGGER TRIGGER_NAME
ON TABLE_NAME
FOR { INSERT | UPDATE | DELETE [, ..] }
AS
SQL_STATEMENTS
[ RETURN ]
```

Oracle的基本语法是：

```
CREATE [ OR REPLACE ] TRIGGER TRIGGER_NAME
[ BEFORE | AFTER ]
[ DELETE | INSERT | UPDATE ]
ON [ USER.TABLE_NAME ]
[ FOR EACH ROW ]
[ WHEN CONDITION ]
[ PL/SQL BLOCK ]
```

下面是使用Oracle语法编写的一个触发器范例：

```

CREATE TRIGGER EMP_PAY_TRIG
AFTER UPDATE ON EMPLOYEE_PAY_TBL
FOR EACH ROW
BEGIN
    INSERT INTO EMPLOYEE_PAY_HISTORY
    (EMP_ID, PREV_PAY_RATE, PAY_RATE, DATE_LAST_RAISE,
    TRANSACTION_TYPE)
    VALUES
    (:NEW.EMP_ID, :OLD.PAY_RATE, :NEW.PAY_RATE,
    :NEW.DATE_LAST_RAISE, 'PAY CHANGE');
END;
/
Trigger created.

```

前面的范例创建了一个名为EMP\_PAY\_TRIG的触发器，每当表EMPLOYEE\_PAY\_TBL里的记录被更新时，它就会在表EMPLOYEE\_PAY\_HISTORY里插入一条记录。

注意：触发器的内容不能修改

触发器的内容是不能修改的。想要修改触发器，我们就只能替换它或重新创建它。有些实现允许使用CREATE TRIGGER语句替换已经存在的同名触发器。

### [22.3.2 DROP TRIGGER语句](#)

这个语句可以删除触发器，其语法如下：

```
DROP TRIGGER TRIGGER_NAME
```

### [22.3.3 FOR EACH ROW语句](#)

MySQL里的触发器还可以调整触发条件。FOR EACH ROW语法可以让过程在SQL语句影响每条记录时都触发，或是一条语句只触发一次。其语法如下所示：

```
CREATE TRIGGER TRIGGER_NAME  
ON TABLE_NAME FOR EACH ROW SQL_STATEMENT
```

区别在于触发器执行的次数。如果创建了一个普通触发器，执行了一条会影响100行记录的SQL语句时，触发器只会执行一次。如果创建触发器时使用了FOR EACH ROW语法，并且再次执行同样的SQL语句，触发器就会执行100次，也就是SQL语句影响的每条记录都会触发它。

## [22.4 动态SQL](#)

动态SQL允许程序员或终端用户在运行时创建SQL语句的具体代码，并且把语句传递给数据库。数据库然后就把数据返回到绑定的程序变量里。

为了更好地理解动态SQL，先要来复习一个静态SQL。本书前面介绍的全部都是静态SQL。静态SQL是事先编写好的，不准备进行改变的。虽然静态SQL语句可以保存到文件里以备以后使用，也可以作为存储过程保存在数据库里，但其灵活性还是不能与动态SQL相比。

使用静态SQL语句的一个问题是，虽然我们可以为终端用户提供大量的语句，但依然可能出现不能满足所有用户需要的情况。动态SQL通常被用于专门的查询工具，允许用户随时创建SQL语句，从而满足特定情况下的特定查询需求。在语句根据用户需要被生成之后，它们被送给数据库，数据库检查语法正确性及所需的权限，对语句进行编译。

使用调用级接口可以创建动态SQL，下一小节将介绍调用级接口。

注意：动态SQL的性能不一定好

虽然动态SQL为终端用户提供了更好的灵活性，但其性能不能与存储过程相比，因为后者已经被SQL优化器进行了解析。

## [22.5 调用级接口](#)

调用级接口（CLI）用于把 SQL 代码嵌入到主机程序，比如 ANSI C。程序员应该很熟悉调用级接口的概念，它是把 SQL 嵌入到不同的过程序编程语言的方法之一。在使用调用级接口时，我们只需要根据主机编程语言的规则把 SQL 语句的文本保存到一个变量里，然后利用这个变量就可以在主机程序里执行 SQL 语句。

EXEC SQL 是一个常见的主机编程语言命令，可以在程序里调用 SQL 语句。

下面是支持 CLI 的常见编程语言：

ANSI C;

C#;

VB.NET;

JAVA;

Pascal;

Fortran。

注意：调用级接口的语法因平台而异

使用调用级接口的具体语法请参考所用主机编程语言的文档。调用级编程语言与平台有关。所以，Oracle 与 SQL Server 的调用级接口互不兼容。

## [22.6 使用 SQL 生成 SQL](#)

使用 SQL 生成 SQL 是节省 SQL 语句编写时间的一个好方法。假设数据库里已经有了 100 个用户，我们创建一个新角色 ENABLE，要授予给这 100 个用户。这时不必手工创建 100 个 GRANT 语句，下面的 SQL 语句会生成所需的每一条语句：

```
SELECT 'GRANT ENABLE TO '|| USERNAME||';'  
FROM SYS.DBA_USERS;
```

这个范例使用了Oracle的系统目录视图（包含着关于用户的信息）。

注意包围GRANT ENABLE TO的单引号，它表示所包围的内容（包括空格在内）要直义使用。还记得吗，我们可以像从表里选择字段一样选择直义值。USERNAME 是系统目录表 SYS.DBA\_USERS 里的字段，双管道符号（||）用于连接字段，它把分号连接到用户名之后，从而形成完整的语句。

这个SQL语句的结果是这样的：

```
GRANT ENABLE TO RRPLEW;  
GRANT ENABLE TO RKSTEP;
```

这些结果应该保存到文件里，再发送给数据库。然后数据库执行文件里的每条SQL语句，这样我们就不必输入很多的命令，从而节省了时间与精力。GRANT ENALBE TO USERNAME语句会对数据库里的每个用户重复执行。

在需要编写会重复多次的SQL语句时，我们应该发挥自己的想象力，让SQL为我们完成工作。

## [22.7 直接SQL与嵌入SQL](#)

直接SQL是指从某种形式的交互终端上执行的SQL语句，它的执行结果会直接返回到终端。本书的大部分内容是关于直接SQL的。直接SQL也被称为交互调用或直接调用。

嵌入SQL是在其他程序里使用的SQL代码，这些程序包括Pascal、Fortran、COBOL和C。前面已经介绍过，SQL 代码是通过调用级接口嵌

入到主机编程语言里的。在主机编程语言里，嵌入 SQL 语句通常以 EXEC SQL 开始，以分号结束。当然也有使用其他结束符的，比如 END-EXEC 和右圆括号。

下面是在主机程序（比如 ANSI C）里嵌入 SQL 的范例：

```
{HOST PROGRAMMING COMMANDS}  
EXEC SQL {SQL STATEMENT};  
{MORE HOST PROGRAMMING COMMANDS}
```

## [22.8 窗口表格函数](#)

窗口表格函数可以对表格的一个窗口进行操作，并且基于这个窗口返回一个值。这样就可以计算连续总和、分级和移动平均值等。窗口表格函数的语法如下所示：

```
ARGUMENT OVER ([PARTITION CLAUSE] [ORDER CLAUSE] [FRAME CLAUSE])
```

几乎所有汇总函数都可以作为窗口表格函数，另外还有5个新的窗口表格函数：

```
RANK() OVER;  
DENSE_RANK() OVER;  
PERCENT_RANK() OVER;  
CUME_DIST() OVER;  
ROW_NUMBER() OVER。
```

一般来说，计算个人在一个收入年度里的评分级别是比较困难的，而窗口表格函数可以让这种工作容易一些，比如下面这个 Microsoft SQL Server 范例：

```
SELECT EMP_ID, SALARY, RANK() OVER (PARTITION BY YEAR(DATE_HIRE)  
ORDER BY SALARY DESC) AS RANK_IN_DEPT  
FROM EMPLOYEE_PAY_TBL;
```

不是全部RDBMS实现都支持窗口表格函数，所以在使用这种函数之前请查看具体实现的文档。

## [22.9 使用XML](#)

2003版的ANSI标准里有一个与XML相关功能的部分，从那之后，很多数据库实现都努力至少支持其中的部分功能。举例来说，ANSI标准里有一部分是以XML格式输出查询的结果，SQL Server就通过语句FOR XML提供了这个功能，范例如下：

```
SELECT EMP_ID, HIRE_DATE, SALARY FROM  
EMPLOYEE_TBL FOR XML AUTO
```

XML 功能集里另一个重要特性是能够从 XML 文档或片断里获取信息，MySQL 通过EXTRACTVALUE函数提供了这个功能，它有两个参数，第一个是XML片断，第二个是定位器，用于返回与字符串匹配标记的第一个值。其语法如下所示：

```
ExtractValue([XML Fragment],[locator string])
```

下面的范例使用这个函数从节点a里提取值：

```
SELECT EXTRACTVALUE('<a>Red<///a><b>Blue</b>','/a') as ColorValue;  
ColorValue  
Red
```

关于XML功能的详细情况请参见具体实现的文档。某些实现，例如SQL Server和Oracle，拥有特定的XML数据类型。例如，Oracle的XMLTYPE类型拥有特定的API来处理与XML有关的大部分功能，例如查找和提取数据。Microsoft SQL Server的XML类型允许使用模板来确保输入到列的XML数据的完整性。



## [22.10 小结](#)

本章介绍了一些高级SQL概念，虽然并没有深入讨论，但可以让我们对这些概念有一个基本的了解。首先是光标，它可以把查询的结果传递到内存里的某个位置。当程序里声明了一个光标之后，在访问之前要打开它，然后就可以把光标的内容获取到一个变量里，用于程序进行处理。光标的内容会保存在内存里，直到光标被关闭且内存被重新分配。

接着介绍了存储过程和触发器。存储过程就是保存在数据库里的SQL语句，这些语句（以及其他命令）在数据库里是经过编译的，可以被用户随时执行。存储过程通常比单个SQL语句具有更好的性能。

另外还介绍了动态SQL、用SQL生成SQL、直接SQL与嵌入SQL的不同。动态SQL是用户在运行期间创建的SQL代码，这是与静态SQL的最大区别。

最后，我们还讨论了窗口表格函数和XML，这些是相对比较新的特性，可能不是所有数据库都支持，但还是值得了解一下。这里介绍的一些高级主题可以用于解释第23章中的企业级SQL应用。

## [22.11 问与答](#)

问：存储过程能够调用另一个存储过程吗？

答：是的，被调用的存储过程被称为嵌套的。

问：如何执行一个光标？

答：只需要使用OPEN CURSOR语句，就会把光标的结果发送到特定存储区域。

## [22.12 实践](#)

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在



于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### 22.12.1 测验

1. 触发器能够被修改吗？
2. 当光标被关闭之后，我们能够重用它的名称吗？
3. 当光标被打开之后，使用什么命令获取它的结果？
4. 触发器能够在INSERT、DELECT或UPDATE语句之前或之后执行吗？
5. 在MySQL里使用什么语句从XML片断里获取信息？
6. 为什么Oracle和MySQL不支持针对光标的DEALLOCATE语法？
7. 为什么光标不是基于数据集的操作？

### 22.12.2 练习

1. 参考下面的MySQL命令，编写SQL语句，来返回数据库中所有表的描述信息：

```
SELECT CONCAT('DESCRIBE ',TABLE_NAME,';') FROM TABLES_PRIV;
```

2. 编写一个 SELECT 语句来生成 SQL 代码，统计每个表里的记录数量。（提示：类似于练习1。）
3. 编写一组SQL命令来创建一个光标，返回所有用户及其销售数据。确保在用户所使用的实现中，正确关闭光标并回收资源。

## 第23章 SQL扩展到企业、互联网和内部网

本章的重点包括：

SQL与企业

前台和后台程序

访问远程数据库

SQL与互联网

SQL与内部网

前一章介绍了一些高级SQL概念，它们基于本书前面章节所介绍的内容，并且开始展示SQL的一些实际应用。本章着重于把SQL扩展到企业背后的概念，其中包括SQL应用程序和让企业全部成员都能够使用数据来完成日常工作。

## [23.1 SQL与企业](#)

很多商业公司都为其他企业、顾客和销售商提供数据，比如一个企业可能会向顾客提供关于产品的详细信息，从而希望实现更好的销售。企业雇员的需求也在考虑之列，比如提供关于雇员的特定信息，包括考勤登记、休假计划、培训计划、公司政策等。在数据库被创建之后，顾客和雇员应该可以通过SQL或某种互联网语言访问企业的数据。

### [23.1.1 后台程序](#)

任何应用的核心都是后台程序，它们对于数据库终端用户是透明的，但却是发生一切事情的幕后场所。后台程序包括实际的数据库服务程序、数据源、把程序连接到Web或局域网上远程数据库的中间软件。

确定所要使用的数据库实现通常是移植任何程序的第一步，包括通过局域网（LAN）到企业、到企业自己的内部网，或是到互联网。移植描述了在一个环境里实现一个应用供用户使用的过程。数据库服务程序应该由数据库管理员（DBA）创建，他理解公司的需求与程序的要求。

应用的中间件包括Web服务程序、能够把Web服务程序连接到数据库服务程序的工具。其主要目的是让Web上的程序能够与公司的数据库

进行通信。

### 23.1.2 前台程序

前台程序是应用的组成部分，终端用户通过它进行交互。前台程序可以是现成的商业软件，或是使用第三方工具自己开发的程序。商业软件包括一些使用Web浏览器来展示内容的应用软件。在Web环境下，类似FireFox和IE这样的浏览器经常被用来访问数据库程序。这样，用户不必安装特定软件也可以访问数据库。

注意：应用具有很多不同的层

前台程序简化了终端用户对数据库的操作。底层的数据库、代码和数据库里发生的事件对于用户来说是透明的。前台程序使得用户不必对系统本身非常了解，从而减少了他们的猜测与疑惑。新技术使得程序更加智能化，让用户能够专注于真正与实际工作有关的部分，从而提高了整体的生产力。

目前可以使用的工具是用户友好的、面向对象的，具有图标、向导，支持鼠标的播放操作。用于把程序移植到Web的流行工具包括Borland公司的C++ Builder、IntraBuilder和微软的Visual Studio。其他一些用于在局域网上开发公司级程序的工具还有Powersoft的PowerBuilder、Oracle公司的Oracle Designer和Oracle Forms、微软的Visual Studio、Borland的Delphi。

图23.1展示了数据库应用里的前台程序和后台程序。后台程序位于数据库所在的主机服务器上。后台用户包括开发人员、程序员、DBA、系统管理员和系统分析员。前台程序位于客户计算机，通常就是每个用户的个人电脑。前台用户是前台程序的大量使用人员，包括数据输入员、会计等。终端用户能够通过网络连接（LAN或广域网）访问后台数据库，这是由一些通过网络为前台和后台程序提供连接的中间件（比如ODBC驱动程序）实现的。

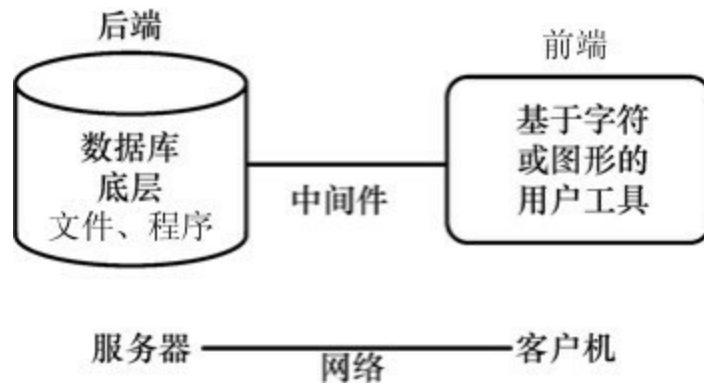


图23.1 数据库应用

## [23.2 访问远程数据库](#)

有时要访问的数据库是个本地数据库，也就是直接连接的。但在很多情况下，我们都会访问某种形式的远程数据库。远程数据库是非本地的，或是说位于非直接连接的服务器上，这时我们必须使用网络和网络协议与数据库进行交互。

访问远程数据库的方式有多种。从广义角度来说，我们是利用中间产品（ODBC和JDBC就是标准的中间件，在后续章节进行介绍）通过网络或互联网连接访问远程数据库的。图23.2展示了访问远程数据库的3种情形。

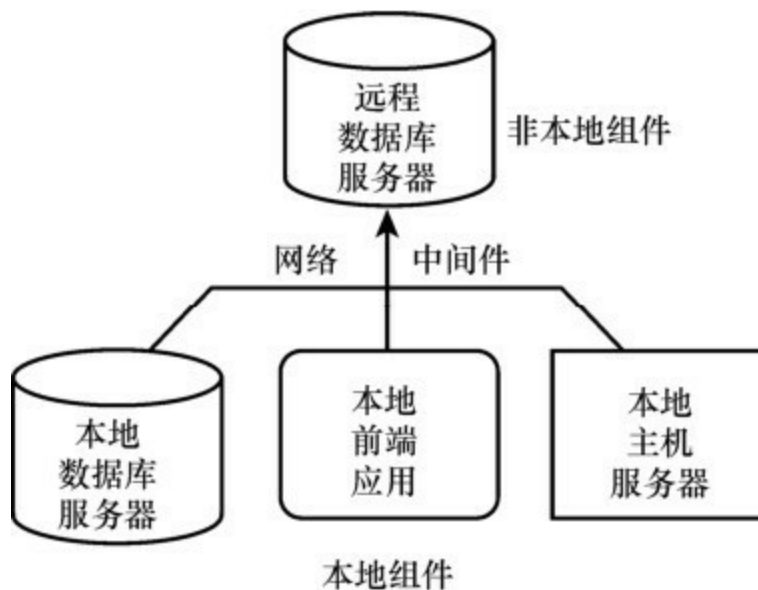


图23.2 访问远程数据库

图23.2展示了从本地数据库服务器、本地前台程序和本地主机服务器访问远程服务器的情形。本地数据库服务器和本地主机服务器经常是同一台机器，因为数据库一般位于本地主机服务器上。但是，我们通常在没有本地数据库连接的情况下从本地服务器连接到远程数据库。对于终端用户来说，前台程序是访问远程数据库的最典型方式。所有的方法都必须把对数据库的请求通过网络进行路由。

### **23.2.1 ODBC**

开放式数据库连接（ODBC）可以通过一个库驱动程序连接到远程数据库。前台程序利用ODBC驱动与后台数据库进行交互。在连接到远程数据库时，可能还需要一个网络驱动。程序调用ODBC函数，驱动管理程序加载ODBC驱动。ODBC驱动处理这个调用，提交SQL请求，从数据库返回结果。

作为 ODBC 的一个组成部分，所有关系数据库管理系统（RDBMS）厂商都提供了数据库的应用编程接口（API）。

### [23.2.2 JDBC](#)

JDBC是Java数据库连接，它类似ODBC，通过一个Java库驱动连接到远程数据库。前台的Java程序使用JDBC驱动与后台的数据库进行交互。

### [23.2.3 OLE DB](#)

OLE DB是微软公司使用组件对象模型（Component Object Model）编写的一组接口，用于代替 ODBC。OLE DB实现力图拓展ODBC功能，不仅可以连接各种数据库实现，&nbsp;也可以连接非数据库存储的数据，例如电子表格等。

除了 ODBC 驱动之外，很多厂商也提供了自己的产品，可以把用户连接到远程数据库。这些厂商产品都是专门用于特定 SQL 实现的，一般不能移植到其他类型的数据库服务程序。

### [23.2.4 厂商连接产品](#)

除了驱动和 API 之外，很多厂商也提供了自己的产品，可以把用户连接到远程数据库。这些厂商产品都是专门用于特定 SQL 实现的，一般不能移植到其他类型的数据库服务程序。

Oracle公司有一个名为Oracle Fusion Middleware的中间件产品，既可以连接Oracle数据库，也可以连接其他应用软件。

Microsoft也有几款产品与其数据库配合使用，例如Microsoft SharePoint Server和 SQL Server Reporting Services。

### [23.2.5 通过Web接口访问远程数据库](#)

通过Web接口访问远程数据库十分类似于通过局域网进行访问，主要区别在于用户的全部请求都经过Web服务程序进行了路由（如图23.3所示）。

从图23.3 中可以看出，一个终端用户通过一个 Web 接口访问数据库，首先是调用一个Web浏览器，它用于连接到一个特定的URL（由Web服务程序的位置决定）。Web服务程序验证用户的访问，把用户请求（可能是一个查询）发送给远程数据库（也可能对用户的身份进行验证）。数据库服务程序然后把结果返回给Web服务程序，后者把结果显示在用户的Web浏览器上。使用防火墙可以控制对特定服务器的非授权访问。

警告：注意互联网信息安全问题

注意在Web上提供的信息。永远要确保在全部恰当的级别都采取了应有的预防措施，其中包括Web服务器、主机服务器、远程数据库。涉及个人隐私的数据，比如个人的社会保险号码，永远都不应该公开在Web上。

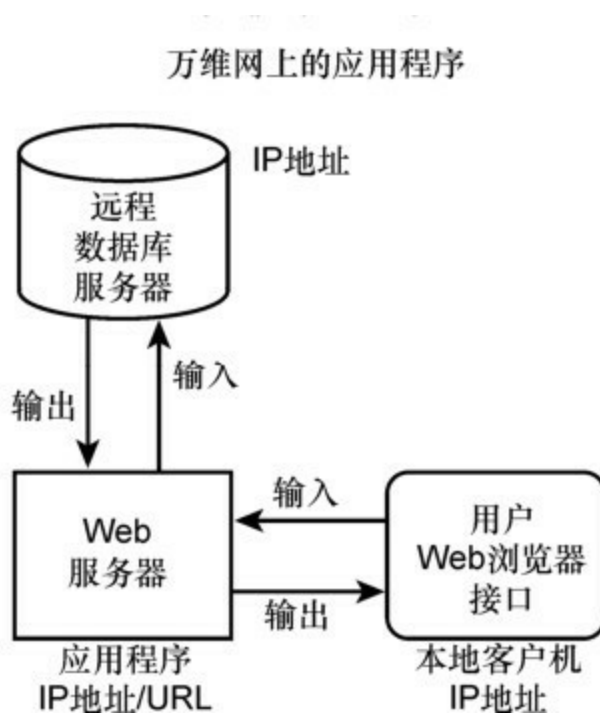


图23.3 远程数据库的Web接口

防火墙是一种安全机制，防止来自和针对服务器的非授权连接。在



一台服务器上可以启动一个或多个防火墙来监视对数据库或服务器的访问。

另外，一些数据库实现允许我们根据IP地址限制对数据库的访问，这就提供了另一层保护，因为我们可以把对数据库的访问限制到充当应用层的Web服务器。

### [23.3 SQL与互联网](#)

SQL可以嵌入到或用于像C#和JAVA这样的编程语言，还可以嵌入到互联网编程语言，比如Java和ASP.NET。源自于HTML的文本可以被转换为SQL，从Web前端远程数据库发送请求。在数据库完成查询操作之后，输出结果被转换回HTML，显示在用户的Web浏览器上。下面的小节将讨论SQL在互联网上的应用。

#### [23.3.1 让数据可以被全世界的顾客使用](#)

随着互联网的出现，数据对全世界的顾客和厂商都开放了。一般来说，用户利用前台工具以只读方式访问数据。

为顾客提供的数据包括一般的顾客信息、产品信息、发票信息、当前订单、延期交货单和其他相关信息。但其中不应该包括隐私信息，比如公司策略和雇员信息。

在互联网上拥有自己的主页已经成为公司竞争中不可缺少的组成部分，Web页面可以仅用很小的代价就向浏览者展示公司的全面情况，包括它的服务、产品和其他信息。

#### [23.3.2 向雇员和授权顾客提供数据](#)

数据库可以通过互联网或公司的内部网向雇员或顾客提供访问。互联网是一个非常有价值的通信资源，可以用于向雇员提供公司政策、福利、培训等信息。但是，在通过互联网提供数据时一定要非常小心，公



司机密和个人信息不应该能够通过Web访问。另外，在线提供的数据库应该只是数据库的一个子集或子集的副本。主要的实用数据库应该全力保护。

警告：互联网的安全性还不够好

与互联网的安全相比，数据库安全更可靠一些，因为后者可以根据所包含的数据进行精细的调整。虽然在通过互联网访问数据时也可以使用一些安全措施，但通常是有限的，而且不像数据库权限那样容易修改。我们应该总是尽量使用数据库服务器具有的安全特性。

### [23.4 SQL与内部网](#)

IBM 最初创建 SQL 是要实现主机上的数据库与使用客户机的用户之间的通信。用户通过 LAN 连接到主机，SQL 被选作数据库与用户之间通信的标准语言。内部网基本上就是一个小型互联网，主要区别是内部网是针对单个公司的应用，而互联网是对公共福斯开放的。内部网上的用户（客户端）接口与客户/服务器环境里的是一样的。SQL经过Web服务器和语言（比如HTML）的路由转发到数据库。内部网主要用于公司内部应用、文档、表单、Web页面和电子邮件。

通过互联网进行的SQL请求必须特别注意性能问题。在这种情况下，不仅需要从数据库获取数据，还需要把数据显示在用户的浏览器上。这通常涉及把数据转换为某种形式的HTML兼容代码。另外，Web连接一般都比内部网连接的速度慢，因此数据来回传递的速度也慢。

连接入Web的数据库实现必须重视安全性。这需要考虑很多问题，来确保数据处于安全保护之下。首先，如果数据暴露于公共网络，必须确保这些数据不会被非法访问。通常，数据会被转换成明文形式，以便用户阅读。可以考虑使用SSL作为部分安全措施，来保护网络交流。SSL使用证书来加密服务端和客户端之间传递的消息，这种加密可以被

用HTTPS开头的网站所识别。

另一个需要考虑的问题是非法的数据输入。用户或应用程序可能会向错误的字段输入了错误的数据类型，也可能会遇到更严重的SQL注入攻击，黑客可能通过这种方式向数据库注入并执行自己的SQL代码。

预防上述问题的最好方法就是，严格约束应用软件账户对数据库的访问。可以在需要访问数据库的时候，使用存储过程和函数，这样就可以对进出系统的数据有所控制。同时，还可以使用户执行任何符合DBA要求的数据操作，以确保数据的一致性。

## 23.5 小结

本章介绍了在互联网上应用SQL和数据库程序背后的概念，这些概念对于公司在当今这个时代保持竞争力是非常重要的。事实已经证明，为了不被时代抛弃，在互联网上占据一席之地是很有好处的——甚至是必须的。为此，公司必须开发程序，甚至是从客户/服务器系统上把程序移植到互联网上的Web服务器。在提供任何类型及任何数量的公司数据时，最需要考虑的问题就是安全，并且应该始终严格坚持安全准则。

本章还讨论了通过局域网和互联网访问远程数据库。任何访问远程数据库的方式都需要使用网络和协议适配器来转换对数据库的请求。在此，我们概要介绍了基于局域网、公司内部网和互联网的SQL应用。在完成后面的测验和练习之后，我们就要进入最后一章了。

## 23.6 问与答

问：为什么说，了解数据是否通过互联网的公共网络被访问，这一点很重要？

答：在客户端和Web应用之间传递的数据往往是明文形式。这就意味着，任何人都可以拦截消息并看到其中的内容，例如社会保险号或银

行账号。在可能的情况下，最好对数据进行加密。

问：针对**Web**应用的后台数据库与针对客户/服务器系统的后台数据库有什么不同吗？

答：针对 **Web** 应用的后台数据库本身不必与针对客户/服务器系统的有什么不同，但基于**Web** 的程序需要满足其他一些要求。举例来说，需要使用**Web** 服务程序访问数据库。在使用**Web**程序时，用户通常不是直接连接到数据库的。

## [23.7 实践](#)

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### [23.7.1 测验](#)

1. 一台服务器上的数据库能够被另一台服务器访问吗？
2. 公司可以使用什么方式向自己的雇员发布信息？
3. 提供数据库连接的产品被称为什么？
4. SQL能够嵌入到互联网编程语言里吗？
5. 如何通过**Web**程序访问远程数据库？

### [23.7.2 练习](#)

1. 连接到互联网，查看一些公司的主页。如果你自己的公司有主页，可以把它与竞争对手的主页进行一下比较，回答以下这些问题。

页面上有动态的内容吗？

什么样的页面或者页面上的什么区域，可能包含来自后端数据库的数据？

Web页面上有什么安全机制吗？在访问保存在数据库里的数据时需要登录吗？

现在，大部分浏览器允许用户查看返回页面的源代码。使用你的网页浏览器查看源代码。其中是否存在一些代码，可以告诉你后端使用的数据库是什么？

如果在源代码中发现了一些信息，例如服务器名称或者数据库用户名，你认为这属于安全漏洞吗？

2. 访问下面的站点，浏览其中的内容、最新的技术和公司在Web上使用的数据（来自于数据库的数据）。

[www.amazon.com](http://www.amazon.com)

[www.informit.com](http://www.informit.com)

[www.epinions.com](http://www.epinions.com)

[www.mysql.com](http://www.mysql.com)

[www.oracle.com](http://www.oracle.com)

[www.ebay.com](http://www.ebay.com)

[www.google.com](http://www.google.com)

## 第24章 标准SQL的扩展

本章的重点包括：

各种实现

不同实现之间的区别

遵循ANSI SQL

交互SQL语句

使用变量

使用参数

本章介绍对ANSI标准SQL的扩展。虽然大多数SQL实现遵循了这个

标准，但有很多厂商会通过各种形式的改进对标准SQL进行扩展。

## [24.1 各种实现](#)

多家厂商发布了多种SQL实现，在此不可能列出全部的关系型数据库厂商，只能讨论一些主流实现，其中包括MySQL、Microsoft SQL Server和Oracle。其他一些比较流行的厂商还有Sybase、IBM、Informix、Progress、PostgreSQL等。

### [24.1.1 不同实现之间的区别](#)

虽然这里讨论的各种实现都是关系型数据库产品，但彼此之间还是有所区别的。这些区别源自于产品设计和数据库发动机处理数据的方式，但本书着重介绍SQL方面的区别。所有的实现都根据ANSI的要求使用SQL作为与数据库通信的语言，但很多实现都对SQL进行了某种形式的扩展。

注意：厂商有意扩展**SQL**标准

不同厂商会出于性能及易用性的考虑对ANSI SQL进行增强，努力提供其他厂商没有的优势，从而吸引顾客。

在了解了SQL之后，根据不同实现的区别对SQL进行调整应该没有什么问题。换句话说，如果我們可以在Sybase实现里编写SQL，就可以在Oracle里编写SQL。另外，了解不同厂商的SQL还可以增加我们的就业机会。

下面比较几个主流厂商与ANSI标准的SELECT语句。

首先是ANSI标准：

```

SELECT [DISTINCT ] [* | COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE SEARCH_CONDITION ]
GROUP BY [ TABLE_ALIAS | COLUMN1 [, COLUMN2 ]
[ HAVING SEARCH_CONDITION ]]
[ ALL ]
[ CORRESPONDING [ BY (COLUMN1 [, COLUMN2 ]) ]
QUERY_SPEC | SELECT * FROM TABLE | TABLE_CONSTRUCTOR ]
[ORDER BY SORT_LIST ]

```

下面是Microsoft SQL Server的语法:

```

[WITH <COMMON_TABLE_EXPRESSION>]
SELECT [DISTINCT][*| COLUMN1 [, COLUMN2, .. ]
[INTO NEW_TABLE]
FROM TABLE1 [, TABLE2 ]
[WHERE SEARCH_CONDITION]
GROUP BY [COLUMN1, COLUMN2,... ]
[HAVING SEARCH_CONDITION]
[ {UNION | INTERSECT | EXCEPT} ][ ALL ]
[ ORDER BY SORT_LIST ]
[ OPTION QUERY_HINT ]

```

Oracle的语法:

```

SELECT [ ALL | DISTINCT ] COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE SEARCH_CONDITION ]
[[ START WITH SEARCH_CONDITION ]

CONNECT BY SEARCH_CONDITION ]
[ GROUP BY COLUMN1 [, COLUMN2 ]
[ HAVING SEARCH_CONDITION ]]
[{UNION [ ALL ] | INTERSECT | MINUS} QUERY_SPEC ]
[ ORDER BY COLUMN1 [, COLUMN2 ]]
[ NOWAIT ]

```

从这些语法的比较可以看出，它们基本上是相同的。它们都具有

SELECT、FROM、WHERE、GROUP BY、HAVING、UNION和ORDER BY子句，这些子句在工作概念上是一样的，但有些具有额外的选项，这些选项就被称为扩展。

### [24.1.2 遵循ANSI SQL](#)

厂商们的确努力遵循ANSI SQL，但都没有做到百分之百符合ANSI SQL标准。有些厂商添加了命令或函数，而且其中很多新命令或函数被吸收到ANSI SQL里。对于厂商来说，遵循标准有很多好处，最明显的是使用其产品易于学习，而且其使用的代码也易于移植到其他实现。当数据库从一个实现迁移到另一个实现时，可移植性是一个非常重要的考虑因素。

对于被认为遵循ANSI的数据库来说，它只需要对应于ANSI标准的一个功能子集。ANSI标准是由多家数据库厂商共同制定的。因此，虽然大多数SQL实现彼此之间有很大差别，但它们都被认为是遵循ANSI标准的。所以，把代码限制到严格遵循ANSI标准的语句能够提高可移植性，但数据库性能可能不会达到最优。总之，我们要在可移植性与性能之间权衡。权衡的结果通常是放弃可移植性，从而充分利用用户所用平台的性能。

### [24.1.3 SQL的扩展](#)

实际上，全部主流厂商都对SQL有所扩展。对于特定实现来说，SQL扩展都是不同的，而且一般不便于移植。然而，流行的标准扩展已经得到了ANSI的关注，将来可能会成为新标准。

Oracle的PL/SQL、Sybase和Microsoft SQL Server使用的Transact-SQL是标准SQL扩展的两个范例，后面的范例里将更详细地介绍它们。

## [24.2 扩展范例](#)



PL/SQL和Transact-SQL都被认为是第4代编程语言，是过程化语言，但SQL是非过程化语言。我们还会简要地讨论一下MySQL。

非过程语言SQL包括如下语句：

INSERT;

UPDATE;

DELETE;

SELECT;

COMMIT;

ROLLBACK。

SQL扩展是一种过程语言，包括标准SQL里全部语句、命令和函数，另外还包括：

变量声明；

光标声明；

条件语句；

循环；

错误处理；

变量累加；

日期转换；

通配符；

触发器；

存储过程。

这些语句可以让程序员在过程化语言里更好地控制数据处理方式。

### **24.2.1 Transact-SQL**

Transact-SQL是Microsoft SQL Server使用的一种过程语言，表示我们告诉数据库如何、在何处获取和操作数据。SQL是非过程的，由数据库决定如何、在何处选择和操作数据。Transact-SQL的几个突出优点包



括声明本地和全局变量、光标、错误处理、触发器、存储过程、循环、通配符、日期转换和汇总报告。

Transact-SQL语句的一个范例如下：

```
IF (SELECT AVG(COST) FROM PRODUCTS_TBL) > 50
BEGIN
    PRINT 'LOWER ALL COSTS BY 10 PERCENT.'
END
ELSE
    PRINT 'COSTS ARE REASONABLE.'
```

这是个很简单的Transact-SQL语句，它表示如果表 PRODUCTS\_TBL里的平均价格大于50，就显示“LOWER ALL COSTS BY 10 PERCENT”，否则就显示“COSTS ARE REASONABLE”。

其中使用了 IF...ELSE 语句计算条件的值，而 PRINT 命令也是个新命令。这些只是Transact-SQL强大功能的九牛一毛。

注意：**SQL**不是过程语言

标准 SQL 从根本上来说是非过程语言，表示我们把语句提交给数据库服务程序，后者决定如何以最优方式执行语句。过程语言允许程序员请求要获取或操作的数据，告诉数据库服务程序如何准确地执行请求。

### [24.2.2 PL/SQL](#)

PL/SQL是Oracle对SQL的扩展，也是一种过程语言，由代码的逻辑块构成。一个逻辑块包含三个部分，其中两个是可选的。第一部分是 DECLARE部分，是可选的。它包含变量、光标和常数。第二个部分是 PROCEDURE，是必需的，包含条件命令和 SQL 语句，是逻辑块的执行部分。第三部分是 EXCEPTION，是可选的，定义了程序如何处理错误和自定义异常。PL/SQL 的突出优点包括使用了变量、常数、光标、属性、循环、处理异常、向程序员显示输出、事务控制、存储过程、触

发器和软件包。

PL/SQL语句的范例如下所示：

```
DECLARE
  CURSOR EMP_CURSOR IS SELECT EMP_ID, LAST_NAME, FIRST_NAME, MIDDLE_NAME
                        FROM EMPLOYEE_TBL;
  EMP_REC EMP_CURSOR%ROWTYPE;
BEGIN
  OPEN EMP_CURSOR;
  LOOP
    FETCH EMP_CURSOR INTO EMP_REC;
    EXIT WHEN EMP_CURSOR%NOTFOUND;
    IF (EMP_REC.MIDDLE_NAME IS NULL) THEN
      UPDATE EMPLOYEE_TBL
      SET MIDDLE_NAME = 'X'
      WHERE EMP_ID = EMP_REC.EMP_ID;
      COMMIT;
    END IF;
  END LOOP;
  CLOSE EMP_CURSOR;
END;
```

这个范例里使用了三个部分里的两个：DECLARE和PROCEDURE。首先，用一个查询定义了一个名为EMP\_CURSOR的光标；然后声明了一个变量EMP\_REC，与光标里每个字段的数据类型（%ROWTYPE）相同。PROCEDURE部分（在BEGIN之后）的第一步是打开光标，然后使用LOOP命令遍历光标里每条记录，结束于END LOOP语句。光标里的全部记录都会更新到表EMPLOYEE\_TBL。如果雇员的中间名是NULL，更新操作会把中间名设置为“X”。更新被提交到数据库，最后光标被关闭。

### **24.2.3 MySQL**

MySQL是个多用户、多线程SQL数据库客户/服务器实现，它包含一个后台服务程序、一个终端监控客户程序、几个客户程序和库。MySQL的主要目标是速度、强健性和易用性，它最初的设计目的是对

大型数据库提供更快速的访问。

MySQL被认为是一种比较符合ANSI标准的数据库实现。从最开始，MySQL就是一个半开源的开发环境，以便严格遵守ANSI标准。从5.0版开始，MySQL推出了开源的社区版和闭源的企业版。2009年，MySQL随同SUN公司一起被Oracle公司收购。

目前，MySQL还不像Oracle或Microsoft SQL Server那样有大的改动，但根据其近期的表现来看，情况很快就会有变化了。用户可以查看所用版本MySQL的说明书，以便了解哪些扩展可能会被开发。

### [24.3 交互SQL语句](#)

交互SQL语句会在完全执行之前询问用户变量、参数或某种形式的数据库。假设我们有一个SQL语句是交互的，用于在数据库里创建用户。它会提示我们输入一些信息，比如用户ID、用户名、电话号码等。它可以创建一个或多个用户，而且只需执行一次。否则，我们就需要用CREATE USER语句分别创建每个用户。当然，这个SQL语句还能提示设置权限。并不是全部厂商都具有交互式SQL语句，详细情况请参见具体实现的文档。

交互式SQL语句的另一个优点是可以使用参数。参数是SQL里的变量，位于程序之内。我们可以在运行时向SQL语句传递参数，让用户能够以更灵活的方式执行语句。很多主流实现支持使用这些参数，下面的小节将展示在Oracle和SQL Server里传递参数的范例。

Oracle里可以把参数传递给静态SQL语句，比如：

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME
FROM EMPLOYEE_TBL
WHERE EMP_ID = '&EMP_ID'
```

前面这个SQL语句会提示输入EMP\_ID，然后返回EMP\_ID和对应的

LAST\_NAME、FIRST\_NAME。下面的语句提示我们输入城市和州，返回居住在指定城市和州里的雇员的全部数据。

```
SELECT *  
FROM EMPLOYEE_TBL  
WHERE CITY = '&CITY'  
AND STATE = '&STATE'
```

在Microsoft SQL Server里，我们可以把参数传递给存储过程：

```
CREATE PROC EMP_SEARCH  
(@EMP_ID)  
AS  
SELECT LAST_NAME, FIRST_NAME  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = @EMP_ID
```

下面就执行这个存储过程并传递参数：

```
SP_EMP_SEARCH "443679012"
```

## [24.4 小结](#)

本章介绍了一些厂商对标准SQL的扩展以及它们遵循ANSI标准的情况。在学习了SQL之后，我们可以轻松地把这些知识（和代码）应用到SQL的其他实现。SQL在不同厂商之间是可以移植的，大多数SQL代码只需要很小的修改就可以在大多数SQL实现中使用。

最后一部分内容展示了三种实现使用的两个扩展。Microsoft SQL Server和Sybase使用了Transact-SQL，而Oracle使用的是PL/SQL。从范例中可以看出这两者之间的相似之处。它们都遵循ANSI标准，在此基础上进行增强，提供更好的功能和效率。另外还介绍了MySQL，其设计目的是提高大型数据库查询的速度。本章的目标是让用户了解到存在着

很多 SQL 扩展，而遵循ANSI SQL标准也是一件非常重要的事情。

如果可以掌握本书的内容并使用它（创建自己的代码、进行测试、增长知识），我们就走上了掌握SQL的阳光大道。公司都要使用数据，没有数据库就很难正常运行。关系型数据库遍布四方，而SQL是与关系型数据库进行通信和管理的标准语言，所以学习SQL是个非常好的选择。祝你好运！

## [24.5 问与答](#)

问：为什么**SQL**有差异？

答：不同的SQL实现使用不同方式存储数据，各个厂商都努力超越其他竞争对手，不断出现的新概念，这些原因导致了SQL有差异。

问：在学习了基本**SQL**之后，我们是不是就可以在不同实现上使用SQL了？

答：是的，但是要记住不同实现之间存在的差异与变化，但大多数实现的SQL基本构架是一样的。

## [24.6 实践](#)

下面的内容包含一些测试问题和实战练习。这些测试问题的目的在于检验对学习内容的理解程度。实战练习有助于把学习的内容应用于实践，并且巩固对知识的掌握。在继续学习之前请先完成测试与练习，答案请见附录C。

### [24.6.1 测验](#)

1. SQL是过程语言还是非过程语言？
2. 除了声明光标之外，光标的3个基本操作是什么？
3. 过程或非过程：数据库发动机在处理什么语句时会决定对SQL

语句进行估值和执行？

### 24.6.2 练习

研究一下不同厂商的SQL差异。访问如下站点，研究常见的SQL实现：

[www.oracle.com](http://www.oracle.com)

[www.sybase.com](http://www.sybase.com)

[www.microsoft.com](http://www.microsoft.com)

[www.mysql.com](http://www.mysql.com)

[www.informix.com](http://www.informix.com)

[www.pgsql.com](http://www.pgsql.com)

[www.ibm.com](http://www.ibm.com)

## [第九部分 附录](#)

附录A 常用SQL命令

附录B 使用数据库进行练习

附录C 测验和练习的答案

附录D 本书范例的CREATE TABLE语句

附录E 书中范例所涉数据的 INSERT语句

附录F 额外练习

术语表

### [附录A 常用SQL命令](#)

下面详细介绍最常用的SQL命令。正像本书中反复强调的，由于很多语句在不同实现中是有区别的，所以它们的详细情况请参见具体实现的文档。

#### [A.1 SQL语句](#)

##### **ALTER TABLE**

```
ALTER TABLE TABLE_NAME
[MODIFY | ADD | DROP]
  [COLUMN COLUMN_NAME][DATATYPE|NULL NOT NULL] [RESTRICT|CASCADE]
[ADD | DROP] CONSTRAINT CONSTRAINT_NAME
```

描述：修改表格的字段。

##### **COMMIT**

```
COMMIT [ TRANSACTION ]
```

描述：把事务保存到数据库。

## **CREATE INDEX**

```
CREATE INDEX INDEX_NAME  
ON TABLE_NAME (COLUMN_NAME)
```

描述：创建表格上的一个索引。

## **CREATE ROLE**

```
CREATE ROLE ROLE_NAME  
[ WITH ADMIN [CURRENT_USER | CURRENT_ROLE]]
```

描述：创建一个数据库角色，它可以被分配一定的系统权限和对象权限。

## **CREATE TABLE**

```
CREATE TABLE TABLE_NAME  
( COLUMN1      DATA_TYPE      [NULL|NOT NULL],  
  COLUMN2      DATA_TYPE      [NULL|NOT NULL])
```

描述：创建数据库的一个表格。

## **CREATE TABLE AS**

```
CREATE TABLE TABLE_NAME AS  
SELECT COLUMN1, COLUMN2,...  
FROM TABLE_NAME  
[ WHERE CONDITIONS ]  
[ GROUP BY COLUMN1, COLUMN2,... ]  
[ HAVING CONDITIONS ]
```

描述：基于数据库的一个表创建另一个表。

## **CREATE TYPE**



```
CREATE TYPE typename AS OBJECT
( COLUMN1      DATA_TYPE      [NULL|NOT NULL],
  COLUMN2      DATA_TYPE      [NULL|NOT NULL])
```

描述：创建自定义类型，可以用于定义表里的字段。

## CREATE USER

```
CREATE USER username IDENTIFIED BY password
```

描述：在数据库中创建一个用户账户。

## CREATE VIEW

```
CREATE VIEW AS
SELECT COLUMN1, COLUMN2,...
FROM TABLE_NAME
[ WHERE CONDITIONS ]
[ GROUP BY COLUMN1, COLUMN2,... ]
[ HAVING CONDITIONS ]
```

描述：创建表格的视图。

## DELETE

```
DELETE
FROM TABLE_NAME
[ WHERE CONDITIONS ]
```

描述：从表格里删除记录。

## DROP INDEX

```
DROP INDEX INDEX_NAME
```

描述：删除表格里的索引。

## DROP TABLE

```
DROP TABLE TABLE_NAME
```

描述：从数据库里删除表。

## **DROP USER**

```
DROP USER user1 [, user2, ...]
```

描述：从数据库里删除用户账户。

## **DROP VIEW**

```
DROP VIEW VIEW_NAME
```

描述：删除表的视图。

## **GRANT**

```
GRANT PRIVILEGE1, PRIVILEGE2, ... TO USER_NAME
```

描述：向用户授予权限。

## **INSERT**

```
INSERT INTO TABLE_NAME [ (COLUMN1, COLUMN2,...)  
VALUES ('VALUE1', 'VALUE2', ...)
```

描述：向表里插入新记录。

## **INSERT...SELECT**

```
INSERT INTO TABLE_NAME  
SELECT COLUMN1, COLUMN2  
FROM TABLE_NAME  
[ WHERE CONDITIONS ]
```

描述：基于一个表向另一个表插入新记录。

## **REVOKE**

```
REVOKE PRIVILEGE1, PRIVILEGE2, ... FROM USER_NAME
```

描述：撤销用户的权限。

## ROLLBACK

```
ROLLBACK [ TO SAVEPOINT_NAME ]
```

描述：撤销数据库事务。

## SAVEPOINT

```
SAVEPOINT SAVEPOINT_NAME
```

描述：创建事务保存点以备回退。

## SELECT

```
SELECT [ DISTINCT ] COLUMN1, COLUMN2, ...  
FROM TABLE1, TABLE2, ...  
[ WHERE CONDITIONS ]  
[ GROUP BY COLUMN1, COLUMN2, ... ]  
[ HAVING CONDITIONS ]  
[ ORDER BY COLUMN1, COLUMN2, ... ]
```

描述：从一个或多个表返回数据，用于创建查询。

## UPDATE

```
UPDATE TABLE_NAME  
SET COLUMN1 = 'VALUE1',  
    COLUMN2 = 'VALUE2', ...  
[ WHERE CONDITIONS ]
```

描述：更新表里的已有数据。

## [A.2 SQL子句](#)

## SELECT

```
SELECT *  
SELECT COLUMN1, COLUMN2,...  
SELECT DISTINCT (COLUMN1)  
SELECT COUNT(*)
```

描述：定义要在查询输出里显示的字段。

## FROM

```
FROM TABLE1, TABLE2, TABLE3,...
```

描述：定义要获取数据的表。

## WHERE

```
WHERE COLUMN1 = 'VALUE1'  
    AND COLUMN2 = 'VALUE2'  
...  
WHERE COLUMN1 = 'VALUE1'  
    OR COLUMN2 = 'VALUE2'  
...  
WHERE COLUMN IN ('VALUE1' [, 'VALUE2'] )
```

描述：定义查询里限制返回数据的条件。

## GROUP BY

```
GROUP BY GROUP_COLUMN1, GROUP_COLUMN2,...
```

描述：排序操作的一种形式，用于把输出划分为逻辑组。

## HAVING

```
HAVING GROUP_COLUMN1 = 'VALUE1'  
    AND GROUP_COLUMN2 = 'VALUE2'  
...
```

描述：类似于WHERE子句，用于在GROUP BY子句里设置条件。

## ORDER BY

```
ORDER BY COLUMN1, COLUMN2, ...  
ORDER BY 1, 2, ...
```

描述：用于对查询结果进行排序。

## 附录B 使用数据库进行练习

本附录里包含了在Windows操作系统里安装MySQL、Microsoft SQL Server和Oracle的指令。MySQL还可以运行于MacOS和Linux。书中提供的指令在本书英文版出版时是正确的，但作者和出版社都不负责软件的授权或提供软件支持。如果遇到了安装问题，或是想获得软件支持，请查看相应实现的说明文档或联系客户支持。

注意：**MySQL**安装说明

用户可能需要查看MySQL的说明文档。访问  
<http://www.mysql.com>，在产品目录下即可找到在线文档链接。

### B.1 在Windows操作系统中安装MySQL的指令

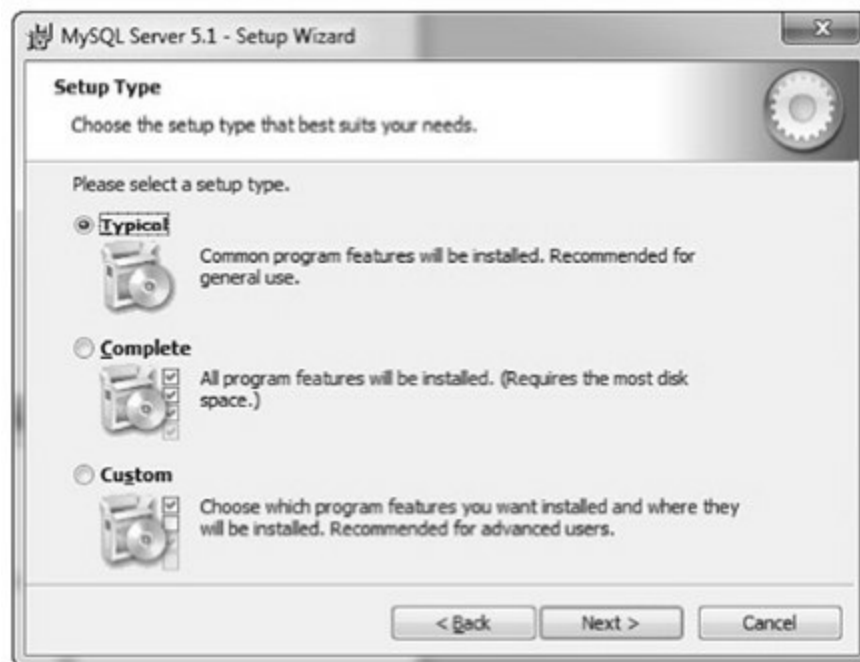
下面的指令用于在微软Windows操作系统上安装MySQL：

1. 从<http://www.mysql.com>下载MySQL.ZIP，这类程序需要使用WinZip或类似的程序对下载的文件进行解压。
2. 在网站上选择Downloads（下载）标签。
3. 选择最新的稳定版本，本书英文版出版时是MySQL Community Server 5.5.8。选择合适的Windows系统插件（msi）文件，并下载到用户的计算机上。
4. 双击msi文件进入安装程序。在欢迎界面，单击“Next”按钮，如图B.1所示。



图B.1 MySQL 安装程序的欢迎界面

5. 在图B.2所示的界面上，选择“Typical”单选项，然后单击“Next”按钮。



图B.2 MySQL 安装程序的选项

6. 在接下来的界面中，单击“Install”按钮，开始安装。
7. 安装成功后，单击“Next”按钮，关闭安装向导。
8. 在图B.3 所示的安装完成界面上，可以勾选复选框以便配置安装实例。之后，单击“Finish”按钮。使用配置向导要比手动配置简单很多。



图B.3 MySQL 安装程序的完成界面

9. 在MySQL的实例配置向导界面上，单击“Next”按钮。
10. 可以选择相应的选项来配置实例，然后单击“Next”按钮。实例配置选项会创建一个新的实例。
11. 此处选择标准配置，然后单击“Next”按钮。
12. 确保在Windows系统设置中包含MySQL的安装路径，然后单击“Next”按钮。这样，即使用户不知道MySQL的安装路径，也可以使用命令行来运行MySQL。
13. 检查安全设置。输入一个 root（管理员）密码，然后单

击“Next”按钮，如图B.4所示。

14. 单击“Execute”按钮，使设置生效。

如果前面的步骤都成功完成，就可以在本书的练习里使用MySQL了。

如果在安装过程中遇到了问题，就卸载MySQL并重复第1步到第14步。如果仍然不能获得或安装MySQL，请联系MySQL来获取帮助。



图B.4 MySQL的安全配置

## [B.2 在Windows操作系统中安装Oracle的指令](#)

下面的指令用于在微软Windows操作系统上安装Oracle:

注意: **Oracle**安装说明

用户可能需要查看 Oracle 的说明文档。访问

<http://www.oracle.com>，在产品和服务目录下即可找到在线文档链接。

1. 进入<http://www.oracle.com>，在下载区域下载合适的安装文件。



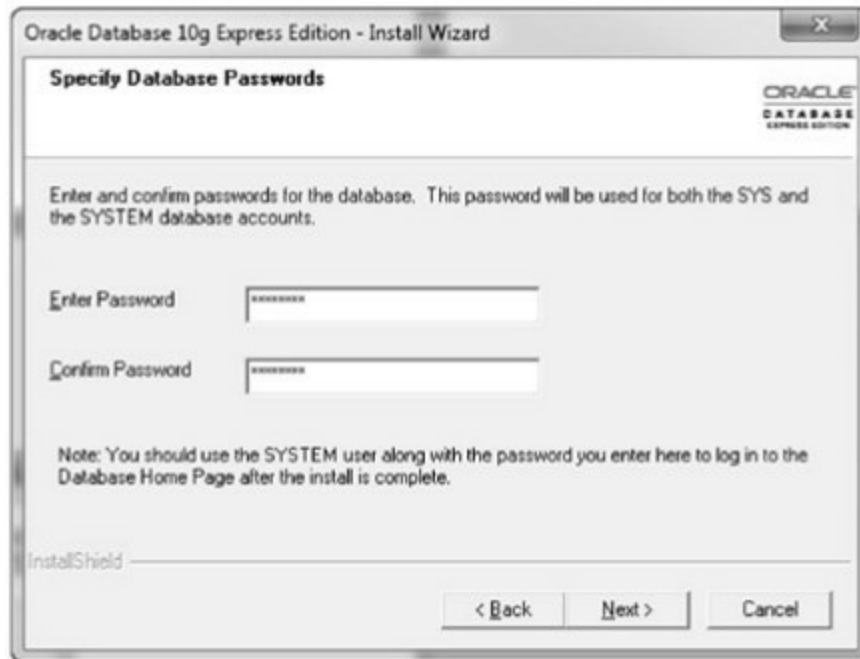
本书所涉范例需要使用Oracle 10g Express Edition版本来执行，这是个免费版本。

2. 双击安装文件进入安装程序，在第一个界面上单击“Next”按钮。
3. 单击同意许可协议，然后单击“Next”按钮。
4. 在图B.5所示界面上，选择默认安装，并选择安装路径，然后单击“Next”按钮。



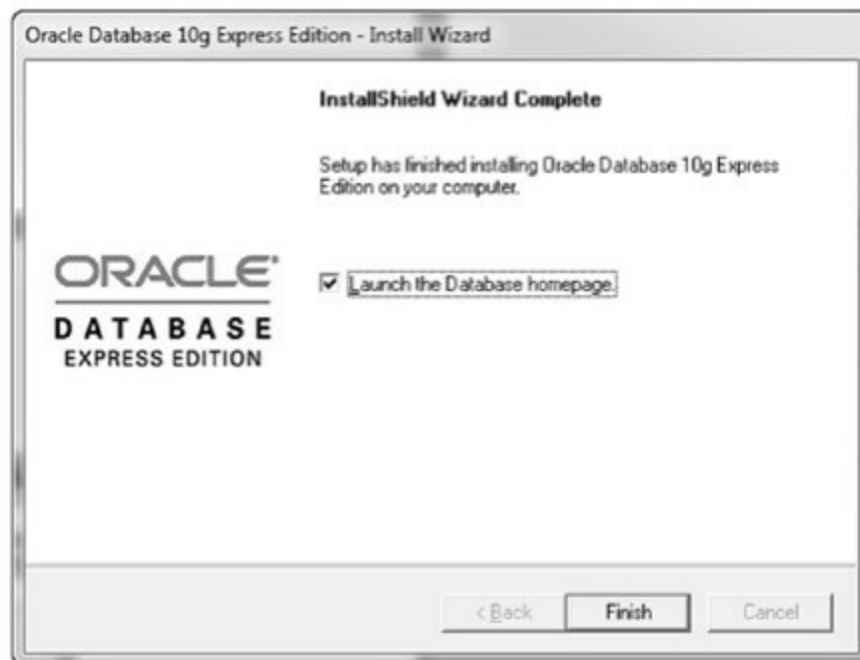
图B.5 Oracle安装路径

5. 输入系统（管理员）密码，如图B.6所示，然后单击“Next”按钮。
6. 在接下来的界面中，单击“Install”按钮，开始安装。



图B.6 设置系统密码

如果安装成功，将会看到图B.7所示的完成界面。



图B.7 Oracle 安装完成界面

如果前面的步骤都成功完成，就可以在本书的练习里使用Oracle了。

如果在安装过程中遇到了问题，就卸载Oracle并重复第1步到第6步。如果仍然不能获得或安装Oracle，请联系Oracle来获取帮助。

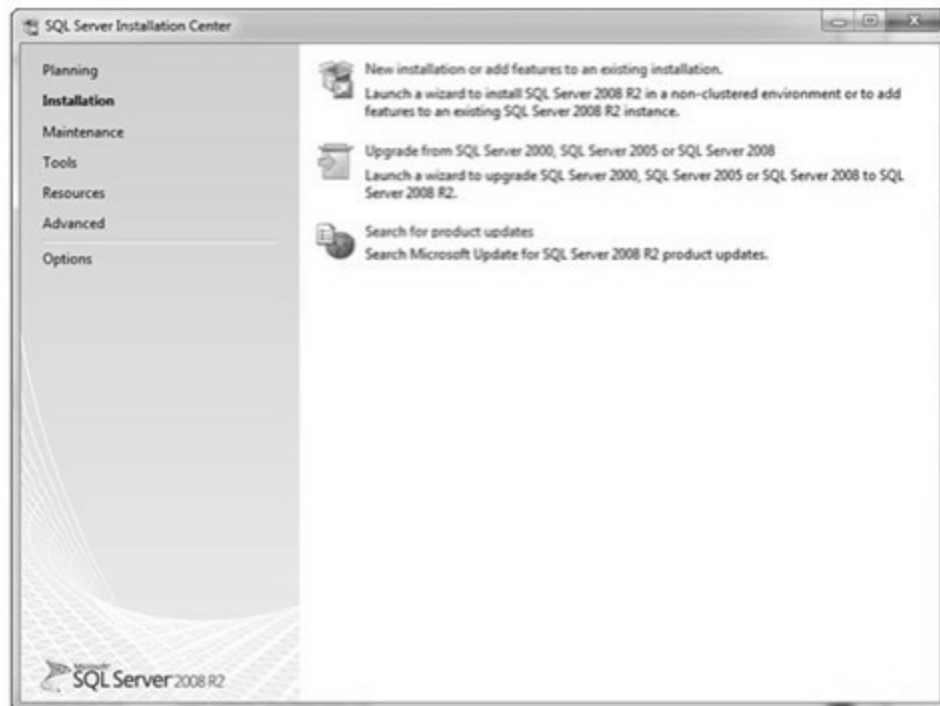
### **B.3 在Windows操作系统中安装Microsoft SQL Server的指令**

下面的指令用于在微软Windows操作系统上安装Microsoft SQL Server:

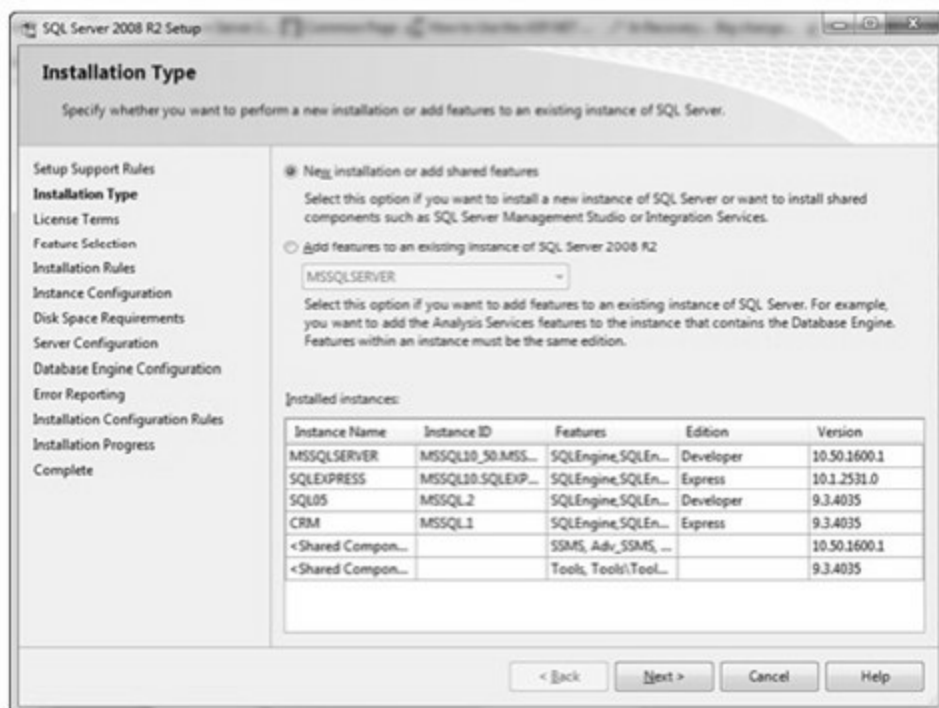
1. 进入[www.microsoft.com/sqlserver/2008/en/us/express.aspx](http://www.microsoft.com/sqlserver/2008/en/us/express.aspx)，单击“Download”，选择合适的安装文件并下载。
2. 双击安装文件，进入图B.8所示的初始化界面。
3. 在右侧的区域选择新的安装选项，如图B.9所示，开始安装将在主安装程序中使用的支持文件。

**注意：Microsoft SQL Server安装说明**

用户可能需要查看Microsoft SQL Server的说明文档。访问[www.microsoft.com/sqlserver/2008/en/us/default.aspx](http://www.microsoft.com/sqlserver/2008/en/us/default.aspx)，在产品信息标签下即可找到在线文档链接。



图B.8 SQL Server 初始化安装界面



图B.9 SQL Server 安装选项界面

4. 选择全新安装，然后单击“Next”按钮。
  5. 接受许可条款，然后单击“Next”按钮。
  6. 选中所有的选项，然后单击“Next”按钮。
  7. 选择默认实例，然后单击“Next”按钮。
  8. 在所需磁盘空间界面单击“Next”按钮。
  9. 在数据库发动机配置界面，单击“Add Current User”按钮，将用户添加为实例管理员，然后单击“Next”按钮。
  10. 在错误报告界面单击“Next”按钮。
  11. 在安装配置规则界面单击“Next”按钮，开始进行安装。
- 如果前面的步骤都成功完成，将会看到一个完成界面。之后就可以在本书的练习里使用Microsoft SQL Server了。
- 如果在安装过程中遇到了问题，就卸载SQL Server并重复第 1步到第 11步。如果仍然不能获得或安装Microsoft SQL Server，请联系Microsoft来获取帮助。

## [附录C 测验和练习的答案](#)

### 第1章

#### 测验答案

1. 缩写“SQL”的含义是什么？  
SQL表示结构化查询语言。
2. SQL命令的6个主要类别是什么？  
数据定义语言（DDL）  
数据操作语言（DML）  
数据查询语言（DQL）  
数据控制语言（DCL）  
数据管理命令（DAC）

事务控制命令（TCC）

3. 4个事务控制命令是什么？

```
COMMIT  
ROLLBACK  
SAVEPOINT  
SET TRANSACTIONS
```

4. 对于数据库访问来说，客户端/服务器模型与Web技术之间的主要区别是什么？

主要区别在于与数据库的连接。使用客户端连接会登录到服务器，直接连接到数据库；而使用Web时，我们会登录到能够到达数据库的互联网上。

5. 如果一个字段被定义为NULL，这是否表示这个字段必须要输入某些内容？

不是。如果某个字段被定义为 NULL，表示字段可以不必输入任何内容。如果字段被定义为NOT NULL，则表示字段必须输入数据。

练习答案

1. 说明下面的SQL命令分别属于哪个类别：

```
CREATE TABLE  
DELETE  
SELECT  
INSERT  
ALTER TABLE  
UPDATE
```

CREATE TABLE: DDL，数据定义语言

DELETE: DML，数据操作语言

SELECT: DQL，数据查询语言

INSERT: DML，数据操作语言

ALTER TABLE: DDL, 数据定义语言

UPDATE: DML, 数据操作语言

2. 观察下面这个表, 选出适合作为主键的列:

EMPLOYEE_TBL	INVENTORY_TBL	EQUIPMENT_TBL
name	item	model
phone	description	year
start date	quantity	serial number
address	item number	equipment number
employee number	location assigned to	

表EMPLOYEE\_TBL的主键应该是雇员号码。每个雇员都会被分配一个唯一的号码, 但雇员可能会有相同的姓名、电话号码、雇佣日期和地址。

表INVENTORY\_TBL的主键应该是物品号码, 其他字段可能包含重复数据。

表EQUIPMENT\_TBL的主键应该是设备号码, 其他字段可能包含重复数据。

3. 不需要答案。

## 第2章

### 测验答案

1. 判断对错: 个人社会保险号码, 输入格式为 '111111111', 它可以是下面任何一种数据类型: 定长字符、变长字符、数值。

答: 对, 只要有效数字达到必要长度。

2. 判断对错: 数值类型的标度是指数值的总体长度。

答: 错。有效数字才是总体长度, 而标度表示小数点右侧保留的位数。

3. 所有的SQL实现都使用同样的数据类型吗?

答: 不是。大多数实现的数据类型都有所不同。虽然它们都遵循

ANSI描述的标准，但不同厂商采取了不同的存储方式，可能导致数据类型有所差异。

4. 下面定义的有效位数和标度分别是多少？

DECIMAL(4,2)  
DECIMAL(10,2)  
DECIMAL(14,1)

答：DECIMAL(4,2)有效位数是4，标度是2；

DECIMAL(10,2)有效位数是10，标度是2；

DECIMAL(14,1)有效位数是14，标度是1。

5. 下面哪个数值能够输入到定义为DECIMAL(4,1)的字段里？

- A. 16.2
- B. 116.2
- C. 16.21
- D. 1116.2
- E. 1116.21

答：前3个数值可以，但16.21会被四舍五入为16.2。数值1116.2和1116.21超过了最多有效位数的限制（4）。

6. 什么是数据？

答：数据是信息的集合，以某种数据类型保存在数据库里。

练习答案

1. 考虑以下字段名称，为它们设置适当的数据类型，确定恰当的长度，并给出一些示范数据：

- a) ssn
- b) state
- c) city
- d) phone\_number



- e) zip
- f) last\_name
- g) first\_name
- h) middle\_name
- i) salary
- j) hourly\_pay\_rate
- k) date\_hired

答: SSN, 定长字符串, '11111111';

STATE, 变长字符串, 'INDIANA';

CITY, 变长字符串, 'INDIANAPOLIS';

PHONE\_NUMBER, 定长字符串, '(555)555-5555';

ZIP, 定长字符串, '46113';

LAST\_NAME, 变长字符串, 'JONES';

FIRST\_NAME, 变长字符串, 'JACQUELINE';

MIDDLE\_NAME, 变长字符串, 'OLIVIA';

SALARY, 数值, 30000;

HOURLY\_PAY\_RATE, 小数, 35.00;

DATE\_HIRED, 日期, '29/10/2007'

2. 同样是这些字段, 判断它们应该是NULL或NOT NULL。体会在不同的应用场合, 有些一般是NOT NULL的字段可能应该是NULL, 反之亦然。

- a) ssn
- b) state
- c) city
- d) phone\_number
- e) zip
- f) last\_name

- g) first\_name
- h) middle\_name
- i) salary
- j) hourly\_pay\_rate
- k) date\_hired

```
SSN—NOT NULL  
STATE—NOT NULL  
CITY—NOT NULL  
PHONE_NUMBER—NULL  
ZIP—NOT NULL  
LAST_NAME—NOT NULL  
FIRST_NAME—NOT NULL  
MIDDLE_NAME—NULL  
SALARY—NULL  
HOURLY_PAY_RATE—NULL  
DATE_HIRED—NOT NULL
```

不是每个人都有电话号码（虽然可能性不大），不是每个人都有中间名，所以这些字段应该允许包含NULL。另外，不是全部雇员都按小时支付工资。

3. 不需要答案。

### 第3章

#### 测验答案

1. 下面这个CREATE TABLE命令能够正常执行吗？需要做什么修改？在不同的数据库（MySQL、Oracle、SQL Server）中执行，有什么限制吗？

```
CREATE TABLE EMPLOYEE_TABLE AS:
( SSN          NUMBER(9)      NOT NULL,
  LAST_NAME    VARCHAR2(20)   NOT NULL,
  FIRST_NAME   VARCHAR(20)    NOT NULL,
  MIDDLE_NAME  VARCHAR2(20)   NOT NULL,
  ST ADDRESS   VARCHAR2(20)   NOT NULL,
  CITY         CHAR(20)       NOT NULL,
  STATE        CHAR(2)        NOT NULL,
  ZIP          NUMBER(4)      NOT NULL,
  DATE HIRED   DATE);
```

答：这个CREATE TABLE语句不能正常执行，语法中有几处错误。下面是正确的语句，适用于Oracle数据库，之后是错误列表。

```
CREATE TABLE EMPLOYEE_TABLE
( SSN          NUMBER()      NOT NULL,
  LAST_NAME    VARCHAR2(20)   NOT NULL,
  FIRST_NAME   VARCHAR2(20)   NOT NULL,
  MIDDLE_NAME  VARCHAR2(20),
  ST_ADDRESS   VARCHAR2(30)   NOT NULL,
  CITY         VARCHAR2(20)   NOT NULL,
  STATE        CHAR(2)        NOT NULL,
  ZIP          NUMBER(5)      NOT NULL,
  DATE_HIRED   DATE );
```

需要修改的是：

- (1) 其中的AS，它不应该出现在这个CREATE TABLE语句里。
- (2) LAST\_NAME字段的NOT NULL之后少了一个逗号。
- (3) 字段MIDDLE\_NAME应该是NULL，因为不是所有人都有中间名。
- (4) 字段 ST ADDRESS应该是 ST\_ADDRESS。如果分成两个单词，数据库会把 ST当作字段名称，把ADDRESS当作一个数据类型，这当然是无效的。
- (5) CITY字段可以正常工作，但使用数据类型VARCHAR2更好

一些。如果全部城市名称都具有同样的长度，数据类型CHAR也可以。

(6) STATE字段少了一个左圆括号。

(7) ZIP字段的长度应该是5而不是4。

(8) 字段DATE HIRED应该是DATE\_HIRED，也就是用下划线让字段名称成为一个连续的字符串。

2. 能从表里删除一个字段吗？

答：当然。但是，虽然这是一个ANSI标准，但还是应该查看具体实现的文档来了解是否支持这个功能。

3. 在前面的表EMPLOYEE\_TBL里创建一个主键约束应该使用什么语句？

答：

```
ALTER TABLE EMPLOYEE_TBL  
ADD CONSTRAINT EMPLOYEE_PK PRIMARY KEY(SSN);
```

4. 为了让前面的表EMPLOYEE\_TBL里的MIDDLE\_NAME字段可以接受NULL值，应该使用什么语句？

答：

```
ALTER TABLE EMPLOYEE_TBL  
MODIFY MIDDLE_NAME VARCHAR(20), NOT NULL;
```

5. 为了让前面的表 EMPLOYEE\_TBL 里添加的人员记录只能位于纽约州('NY')，应该使用什么语句？

答：

```
ALTER TABLE EMPLOYEE_TBL  
ADD CONSTRAINT CHK_STATE CHECK(STATE='NY');
```

6. 要在前面的表EMPLOYEE\_TBL里添加一个名为EMPID的自动增量字段，应该使用什么语句，才能同时符合MySQL和SQL Server的语法结构？

答：

```
ALTER TABLE EMPLOYEE_TBL  
ADD COLUMN EMPID INT AUTO_INCREMENT;
```

练习答案

不需要答案。

第4章

测验答案

1. 判断正误。规格化是把数据划分为逻辑相关组的过程。

答：对。

2. 判断正误。让数据库里没有重复或冗余数据，让数据库里所有内容都规格化，总是最好的方式。

答：错，不一定。规格化会让更多的表需要结合，增加I/O和CPU时间，从而降低数据库性能。

3. 判断正误。如果数据是第三规格形式，它会自动属于第一和第二规格形式。

答：对。

4. 与规格化数据库相比，反规格化数据库的主要优点是什么？

答：最大优点是改善性能。

5. 反规格化的主要缺点是什么？

答：冗余和重复数据会占据额外的空间，难以编程，需要更多的数据维护工作。

6. 在以数据库进行规格化时，如何决定数据是否需要转移到单独

的表？

答：如果表包含冗余的数据组，这些数据就可以转移到单独的表里。

7. 对数据库设计进行过度规格化的缺点是什么？

答：过度规格化会大量占用CPU和内存资源，给服务器造成很大的压力。

练习答案

1. 为一家小公司开发一个新数据库，使用如下数据，对其进行规格化。记住，即使是一家小公司，其数据库的复杂程度也会超过这里给出的范例。

雇员：

Angela Smith, secretary, 317-545-6789, RR 1 Box 73, Greensburg, Indiana, 47890, \$9.50 hour, date started January 22, 1996, SSN is 323149669.

Jack Lee Nelson, salesman, 3334 N Main St, Brownsburg, IN, 45687, 317-852-9901, salary of \$35,000.00 year, SSN is 312567342, date started 10/28/95.

顾客：

Robert's Games and Things, 5612 Lafayette Rd, Indianapolis, IN, 46224, 317-291-7888, customer ID is 432A.

Reed's Dairy Bar, 4556 W 10th St, Indianapolis, IN, 46245, 317-271-9823, customer ID is 117A.

顾客订单：

Customer ID is 117A, date of last order is February 20, 1999, the product ordered was napkins, and the product ID is 661.

答：

<b>Employees</b>	<b>Customers</b>	<b>Orders</b>
SSN	CUSTOMER ID	CUSTOMER ID
NAME	NAME	PRODUCT ID
STREET ADDRESS	STREET ADDRESS	PRODUCT
CITY	CITY	DATE ORDERED
STATE	STATE	
ZIP ZIP		
PHONE NUMBER	PHONE NUMBER	
SALARY		
HOURLY PAY		
START DATE		
POSITION		

2. 不需要答案。

## 第5章

### 测验答案

使用具有如下结构的表EMPLOYEE\_TBL:

	Column	data type	(not)null
last_name	varchar2(20)	not null	
first_name	varchar2(20)	not null	
ssn	char(9)	not null	
phone	number(10)	null	
LAST_NAME	FIRST_NAME	SSN	PHONE
SMITH	JOHN	312456788	3174549923
ROBERTS	LISA	232118857	3175452321
SMITH	SUE	443221989	3178398712
PIERCE	BILLY	310239856	3176763990

下列语句运行后会有什么结果？

a)

```
INSERT INTO EMPLOYEE_TBL
('JACKSON', 'STEVE', '313546078', '3178523443');
```

答：这个INSERT语句不会运行，因为缺少了关键字VALUES。

b)

```
INSERT INTO EMPLOYEE_TBL VALUES  
( 'JACKSON', 'STEVE', '313546078', '3178523443' );
```

答：一条记录会被插入到表EMPLOYEE\_TBL。

c)

```
INSERT INTO EMPLOYEE_TBL VALUES  
( 'MILLER', 'DANIEL', '230980012', NULL );
```

答：一条记录会被插入到表EMPLOYEE\_TBL，字段PHONE的值是NULL。

d)

```
INSERT INTO EMPLOYEE_TBL VALUES  
( 'TAYLOR', NULL, '445761212', '3179221331' );
```

答：这个 INSERT语句不会执行，因为字段FIRST\_NAME是NOT NULL。

e)

```
DELETE FROM EMPLOYEE_TBL;
```

答：表EMPLOYEE\_TBL里的全部记录都会被删除。

f)

```
DELETE FROM EMPLOYEE_TBL  
WHERE LAST_NAME = 'SMITH';
```

答：表EMPLOYEE\_TBL里全部姓SMITH的记录都会被删除。



g)

```
DELETE FROM EMPLOYEE_TBL  
WHERE LAST_NAME = 'SMITH'  
AND FIRST_NAME = 'JOHN';
```

答：表EMPLOYEE\_TBL里 JOHN SMITH的记录会被删除。

h)

```
UPDATE EMPLOYEE_TBL  
SET LAST_NAME = 'CONRAD';
```

答：所有记录的LAST\_NAME字段都被设置为CONRAD。

i)

```
UPDATE EMPLOYEE_TBL  
SET LAST_NAME = 'CONRAD'  
WHERE LAST_NAME = 'SMITH';
```

答：JOHN SMITH和SUE SMITH现在变成 JOHN CONRAD和SUE CONRAD。

j)

```
UPDATE EMPLOYEE_TBL  
SET LAST_NAME = 'CONRAD',  
FIRST_NAME = 'LARRY';
```

答：全部雇员的姓名现在都是LARRY CONRAD。

k)

```
UPDATE EMPLOYEE_TBL
SET LAST_NAME = 'CONRAD',
FIRST_NAME = 'LARRY'
WHERE SSN = '312456788';
```

答：JOHN SMITH现在变成LARRY CONRAD。

练习答案

1. 不需要答案。
2. 使用表PRODUCTS\_TBL进行下面的练习。
  - a) 向产品表添加如下产品：

PROD_ID	PROD_DESC	COST
301	FIREMAN COSTUME	24.99
302	POLICEMAN COSTUME	24.99
303	KIDDIE GRAB BAG	4.99

答：

```
INSERT INTO PRODUCTS_TBL VALUES
('301','FIREMAN COSTUME',24.99);
INSERT INTO PRODUCTS_TBL VALUES
('302','POLICEMAN COSTUME',24.99);
INSERT INTO PRODUCTS_TBL VALUES
('303','KIDDIE GRAB BAG',4.99);
```

- b) 利用DML修改所添加两种服装的价格，它们应该与女巫的服装同价。

答：

```
UPDATE PRODUCTS_TBL  
SET COST = 29.99  
WHERE PROD_ID = '301';
```

```
UPDATE PRODUCTS_TBL  
SET COST = 29.99  
WHERE PROD_ID = '302';
```

c) 现在要缩减产品线，首先对新产品下手。删除刚刚添加的3种产品。

答：

```
DELETE FROM PRODUCTS_TBL WHERE PROD_ID = '301';  
DELETE FROM PRODUCTS_TBL WHERE PROD_ID = '302';  
DELETE FROM PRODUCTS_TBL WHERE PROD_ID = '303';
```

d) 在执行DELETE语句之前，有什么办法可以用来确定所删除的数据准确无误呢？

答：为了确保所删除的内容准确无误，需要先执行一个SELECT语句，其中的FROM和WHERE子句与删除语句相同。

## 第6章

### 测验答案

1. 判断正误。如果提交了一些事务，还有一些事务没有提交，这时执行ROLLBACK命令，同一过程里的全部事务都会被撤销。

答：错。当事务被提交之后，是不能被回退的。

2. 判断正误。SAVEPOINT命令会把一定数量已执行事务之后的事务保存起来。

答：错。保存点只是回退的一个标记点。

3. 简要叙述下面每个命令的作用：COMMIT、ROLLBACK和SAVEPOINT。

答：COMMIT保存由事务产生的变化。ROLLBACK撤销由事务产

生的变化。SAVEPOINT在事务里创建用于回退的逻辑点。

4. 在Microsoft SQL Server中执行事务有什么不同点？

答：除非将语句置于事务之中，否则SQL Server会自动提交执行语句。此外，SQL Server中的SAVEPOINT语法也不同。同时，SQL Server不支持RELEASE SAVEPOINT命令。

5. 使用事务进行操作的实质是什么？

答：事务会对临时存储空间进行操作，因为数据库服务器需要记录语句执行前的所有变化，以便在需要ROLLBACK的时候进行撤销。

练习答案

1. 执行如下事务，并且在第3个事务之后创建一个保存点或者一个保存事务，然后在最后执行一条ROLLBACK命令。请说明上述操作完成之后表CUSTOMER\_TBL的内容。

答：

```
INSERT INTO CUSTOMER_TBL VALUES(615,'FRED WOLF','109 MEMORY  
LANE','PLAINFIELD','IN',46113,'3175555555',NULL);  
INSERT INTO CUSTOMER_TBL VALUES(559,'RITA THOMPSON',  
'125PEACHTREE','INDIANAPOLIS','IN',46248,'3171111111',NULL);  
INSERT INTO CUSTOMER_TBL VALUES(715,'BOB DIGGLER',  
'1102 HUNTINGTON ST','SHELBY','IN',41234,'3172222222',NULL);  
SAVEPOINT SAVEPOINT1;  
UPDATE CUSTOMER_TBL SET CUST_NAME='FRED WOLF' WHERE  
CUST_ID='559';  
UPDATE CUSTOMER_TBL SET CUST_ADDRESS='APT C 4556 WATERWAY'  
WHERE CUST_ID='615';  
UPDATE CUSTOMER_TBL SET CUST_CITY='CHICAGO' WHERE CUST_ID='715';  
ROLLBACK;
```

2. 执行如下事务，在第3个事务之后创建一个保存点。

事务执行完之后添加一条COMMIT命令，之后再加上一条回退到保存点的ROLLBACK命令，这时会发生什么呢？

答：

```
UPDATE CUSTOMER_TBL SET CUST_NAME='FRED WOLF' WHERE  
CUST_ID='559';  
UPDATE CUSTOMER_TBL SET CUST_ADDRESS='APT C 4556 WATERWAY'  
WHERE CUST_ID='615';  
UPDATE CUSTOMER_TBL SET CUST_CITY='CHICAGO' WHERE CUST_ID='715';  
SAVEPOINT SAVEPOINT1;  
DELETE FROM CUSTOMER_TBL WHERE CUST_ID='615';  
DELETE FROM CUSTOMER_TBL WHERE CUST_ID='559';  
DELETE FROM CUSTOMER_TBL WHERE CUST_ID='615';  
COMMIT;  
ROLLBACK;
```

由于语句已经被提交了，所以ROLLBACK语句没有任何效果。

## 第7章

### 测验答案

1. 说出任何SELECT语句都需要的组成部分。

答：SELECT和FROM关键字，或称为子句，是所有SELECT语句都必须具有的。

2. 在WHERE子句里，任何数据都需要使用单引号吗？

答：不是。字符数据类型需要使用单引号，数值数据不需要。

3. SELECT语句属于SQL语言里哪一类命令？

答：SELECT语句属于数据查询语言。

4. WHERE子句里能使用多个条件吗？

答：可以。SELECT、INSERT、UPDATE和DELETE语句里的WHERE子句可以包含多个条件，这些条件是通过操作符AND和OR关联在一起的，详情请见第8章。

5. DISTINCT选项的作用是什么？

答：使用这个选项时不会显示重复内容。

6. 选项ALL是必需的吗？

答：不是。虽然这个选项是可以使用的，但它不是必需的。

7. 在基于字符字段进行排序时，数字字符是如何处理的？

答：它们按照 ASCII 字符进行排序，这意味着数字字符会这样排序：1、12、2、222、22222、3、33。

8. 在大小写敏感性方面，Oracle与MySQL和Microsoft SQL Server有什么不同？

答：Oracle默认是大小写敏感的。

练习答案

1. 在计算机上运行RDBMS。使用数据库learnsql，输入以下SELECT命令。判断其语法是否正确，如果不正确就进行必要的修改。这里使用的是表EMPLOYEE\_TBL。

a)

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME,  
FROM EMPLOYEE_TBL;
```

答：这个SELECT语句不会执行，因为FIRST\_NAME字段后面多了一个逗号，正确的语法应该是这样的：

a)

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME  
FROM EMPLOYEE_TBL;
```

b)

```
SELECT EMP_ID, LAST_NAME  
ORDER BY EMP_ID  
FROM EMPLOYEE_TBL;
```

答：这个SELECT语句不会执行，因为FROM和ORDER BY子句的次序有误。正确的语法应该是这样的：

```
SELECT EMP_ID, LAST_NAME  
FROM EMPLOYEE_TBL  
ORDER BY EMP_ID;
```

c)

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = '213764555'  
ORDER BY EMP_ID;
```

答：这个SELECT语句是正确的。

d)

```
SELECT EMP_ID SSN, LAST_NAME  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = '213764555'  
ORDER BY 1;
```

答：这个SELECT语句是正确的。注意其中的EMP\_ID字段被重命名为SSN。

e)

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = '213764555'  
ORDER BY 3, 1, 2;
```

答：这个SELECT语句的语法是正确的。注意ORDER BY子句里字段的次序。返回数据的排序先后是：首先是FIRST\_NAME，接着是EMP\_ID，最后是LAST\_NAME。

2. 下面这个SELECT语句能工作吗？

```
SELECT LAST_NAME, FIRST_NAME, PHONE  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = '33333333';
```

答：虽然这个语句不会返回什么数据，但语法是正确的，语句可以执行。没有返回数据是因为没有记录的EMP\_ID是33333333。

3. 编写一条SELECT语句，从表PRODUCTS\_TBL里返回每件产品的名称和价格。哪个产品是最贵的？

答：

```
SELECT PROD_DESC,COST FROM PRODUCTS_TBL;  
The witch costume is the most expensive.
```

4. 编写一个查询，生成全部顾客及其电话号码的列表。

答：

```
SELECT CUST_NAME,CUST_PHONE FROM CUSTOMER_TBL;
```

5. 答案有所不同。

## 第8章

### 测验答案

1. 判断正误：在使用操作符OR时，全部条件都必须是TRUE。

答：错。只需要有一个条件为TRUE。

2. 判断正误：在使用操作符IN时，所有指定的值都必须匹配。

答：错。只需要有一个值匹配。

3. 判断正误：操作符AND可以用于SELECT和WHERE子句。

答：错。操作符AND只能用于WHERE子句。

4. 判断正误：操作符ANY可以使用一个表达式列表。

答：错。操作符ANY不能使用表达式列表。



5. 操作符IN的逻辑求反是什么？

答：NOT IN。

6. 操作符ANY和ALL的逻辑求反是什么？

答：<>ANY和<>ALL。

7. 下面的SELECT语句有错吗？错在何处？

a)

```
SELECT SALARY
FROM EMPLOYEE_PAY_TBL
WHERE SALARY BETWEEN 20000, 30000;
```

答：20000和30000之间少了AND。正确的语法是：

```
SELECT SALARY
FROM EMPLOYEE_PAY_TBL
WHERE SALARY BETWEEN 20000 AND 30000;
```

b)

```
SELECT SALARY + DATE_HIRE
FROM EMPLOYEE_PAY_TBL;
```

答：字段DATE\_HIRE的数据类型是DATE，不能用于算术函数。

c)

```
SELECT SALARY, BONUS
FROM EMPLOYEE_PAY_TBL
WHERE DATE_HIRE BETWEEN 1999-09-22
AND 1999-11-23
AND POSITION = 'SALES'
OR POSITION = 'MARKETING'
AND EMP_ID LIKE '%55%';
```

答：语法正确。

## 练习答案

1. 使用下面这个表CUSTOMER\_TBL，编写一条SELECT语句，选择住在Indiana、Ohio、Michigan和Illinois并且姓名以字母A或B开头的客户，返回它们的ID和姓名（以字母顺序）。

DESCRIBE CUSTOMER_TBL		
Name	Null?	Type
-----		
CUST_ID	NOT NULL	VARCHAR (10)
CUST_NAME	NOT NULL	VARCHAR (30)
CUST_ADDRESS	NOT NULL	VARCHAR (20)
CUST_CITY	NOT NULL	VARCHAR (12)
CUST_STATE	NOT NULL	VARCHAR (2)
CUST_ZIP	NOT NULL	VARCHAR (5)
CUST_PHONE		VARCHAR (10)
CUST_FAX		VARCHAR (10)

答：

```
SELECT CUST_ID, CUST_NAME, CUST_STATE
FROM CUSTOMER_TBL
WHERE CUST_STATE IN ('IN', 'OH', 'MI', 'IL')
AND CUST_NAME LIKE 'A%'
OR CUST_NAME LIKE 'B%'
ORDER BY CUST_NAME;
```

2. 使用下面这个表PRODUCTS\_TBL，编写一个SQL语句，选择产品价格价格在\$1.00与\$12.50之间的产品，返回它们的ID、描述和价格。

DESCRIBE PRODUCTS_TBL		
Name	Null?	Type
-----		
PROD_ID	NOT NULL	VARCHAR (10)
PROD_DESC	NOT NULL	VARCHAR (25)
COST	NOT NULL	DECIMAL(6,2)

答：

```
SELECT *  
FROM PRODUCTS_TBL  
WHERE COST BETWEEN 1.00 AND 12.50;
```

3. 如果在第2个练习题里使用了操作符BETWEEN，重新编写SQL语句，使用另一种操作符来得到相同的结果。如果没有使用BETWEEN，现在就来用一用。

答：

```
SELECT *  
FROM PRODUCTS_TBL  
WHERE COST >= 1.00 AND COST <= 12.50;
```

```
SELECT *  
FROM PRODUCTS_TBL  
WHERE COST BETWEEN 1.00 AND 12.50;
```

4. 编写一个SELECT语句，返回价格小于1.00或大于12.50产品。有两种方法可以实现。

答：

```
SELECT *  
FROM PRODUCTS_TBL  
WHERE COST < 1.00 OR COST > 12.50;
```

```
SELECT *  
FROM PRODUCTS_TBL  
WHERE COST NOT BETWEEN 1.00 AND 12.50;
```

还要注意的，BETWEEN包含上限和下限，而NOT BETWEEN不包含。

5. 编写一个SELECT语句，从表PRODUCTS\_TBL返回以下信息：

产品描述、产品价格、每个产品5%的销售税。产品列表按价格从高到低排列。

答：

```
SELECT PROD_DESC, COST, COST * .05  
FROM PRODUCTS_TBL  
ORDER BY COST DESC;
```

6. 编写一个SELECT语句，从表PRODUCTS\_TBL返回以下信息：产品描述、产品价格、每个产品 5%的销售税、加上销售税的总价。产品列表按价格从高到低排列。有两种方法可以实现。

答：

```
SELECT PROD_DESC, COST, COST * .05, COST + (COST * .05)  
FROM PRODUCTS_TBL  
ORDER BY COST DESC;
```

```
SELECT PROD_DESC, COST, COST * .05, COST * 1.05  
FROM PRODUCTS_TBL  
ORDER BY COST DESC;
```

7. 任选PRODUCTS\_TBL表中的3种产品。编写一个查询，返回这3种产品的相关记录。之后，再重新编写一个查询，返回除这3种产品之外的所有产品记录。在查询中，组合使用相等操作符和连接操作符。

答：

```
SELECT *
  FROM PRODUCTS_TBL
 WHERE PROD_ID=11235
    OR PROD_ID=119
    PROD_ID=13;
```

```
SELECT *
  FROM PRODUCTS_TBL
 WHERE PROD_ID<>11235
    AND PROD_ID<>119
    AND PROD_ID<>13;
```

8. 使用IN操作符重新编写练习题7中的查询。比较两种写法，哪种更高效？哪种更易读？

答：

```
SELECT *      FROM PRODUCT_TBL
 WHERE PROD_ID IN (11235,119,13);
```

```
SELECT *
  FROM PRODUCT_TBL
 WHERE PROD_ID IN (11235,119,13);
```

9. 编写一个查询，返回所有名称以P开头的产品的记录。之后，再重新编写一个查询，返回所有名称不以P开头的产品的记录。

答：

```
SELECT *
  FROM PRODUCTS_TBL
 WHERE PROD_DESC LIKE ('P%');
SELECT *
  FROM PRODUCTS_TBL
 WHERE PROD_DESC NOT LIKE ('P%');
```

## 第9章

## 测验答案

1. 判断正误：AVG函数返回全部行里指定字段的平均值，包括NULL值。

答：错，不会考虑NULL值。

2. 判断正误：SUM函数用于统计字段之和。

答：错，SUM函数用于返回一组记录之和。

3. 判断正误：COUNT(\*)函数统计表里的行数。

答：对。

4. 下面的SELECT语句能运行吗？如果不行，应该如何修改？

a)

```
SELECT COUNT *  
FROM EMPLOYEE_PAY_TBL;
```

答：这个语句不能执行，因为星号两侧少了一对圆括号。正确语法是：

```
SELECT COUNT(*)  
FROM EMPLOYEE_PAY_TBL;
```

b)

```
SELECT COUNT(EMP_ID), SALARY  
FROM EMPLOYEE_PAY_TBL  
GROUP BY SALARY;
```

答：语法正确，语句可以执行。

c)

```
SELECT MIN(BONUS), MAX(SALARY)  
FROM EMPLOYEE_PAY_TBL  
WHERE SALARY > 20000;
```

答：语法正确，语句可以执行。

d)

```
SELECT COUNT(DISTINCT PROD_ID) FROM PRODUCTS_TBL;
```

答：语法正确，语句可以执行。

e)

```
SELECT AVG(LAST_NAME) FROM EMPLOYEE_TBL;
```

答：这个语句不能执行，因为字段LAST\_NAME不是数值数据类型的。

f)

```
SELECT AVG(PAGER) FROM EMPLOYEE_TBL;
```

答：语法正确，语句可以执行。

### 练习答案

1. 利用表EMPLOYEE\_TBL构造SQL语句，完成如下练习。

a) 平均薪水是多少？

答：平均薪水是\$30 000.00。返回这个数据的SQL语句是：

```
SELECT AVG(SALARY)
FROM EMPLOYEE_PAY_TBL;
```

b) 最大奖金是多少？

答：最大资金是\$2 000.00。返回这个数据的SQL语句是：

```
SELECT MAX(BONUS)
FROM EMPLOYEE_PAY_TBL;
```

c) 总薪水是多少？

答：总薪水是\$90 000.00。返回这个数据的SQL语句是：

```
SELECT SUM(SALARY)
FROM EMPLOYEE_PAY_TBL;
```

d) 最低小时工资是多少？

答：最低小时工资是\$11.00，返回这个数据的SQL语句是：

```
SELECT MIN(PAY_RATE)
FROM EMPLOYEE_PAY_TBL;
```

e) 表里有多少行记录？

答：表里的记录总数是6，返回这个数据的SQL语句是：

```
SELECT COUNT(*)
FROM EMPLOYEE_PAY_TBL;
```

2. 有多少雇员的姓以G开头？

答：有两个。获得这个数据的SQL语句是：

```
SELECT COUNT(*)
FROM EMPLOYEE_TBL
WHERE LAST_NAME LIKE 'G%';
```

3. 编写一个查询，来确定系统中所有订单的总额。如果每个产品的价格是\$10.00，全部订单的总额是多少？

答：

```
SELECT SUM(COST*QTY)
FROM ORDERS_TBL,PRODUCTS_TBL
WHERE ORDERS_TBL.PROD_ID=PRODUCTS_TBL.PROD_ID;
SELECT SUM(QTY) * 10
FROM ORDERS_TBL;
```



4. 如果所有雇员的姓名按照字母表排序，那么编写一个查询，来确定第一个和最后一个雇员的姓名是什么？

答：

```
SELECT MIN(LAST_NAME) AS LAST_NAME FROM EMPLOYEE_TBL;
```

```
SELECT MAX(LAST_NAME) AS LAST_NAME  
FROM EMPLOYEE_TBL;
```

5. 编写一个查询，对雇员姓名列使用AVG函数。查询语句能运行吗？思考为什么会产生这样的结果。

答：

```
SELECT AVG(LAST_NAME) AS LAST_NAME FROM EMPLOYEE_TBL;
```

此处会报错，因为这里不是数值型数据。

## 第10章

### 测验答案

1. 下面的SQL语句能正常执行吗？

a)

```
SELECT SUM(SALARY), EMP_ID  
FROM EMPLOYEE_PAY_TBL  
GROUP BY 1 AND 2;
```

答：不能。GROUP BY子句里的 and是不正确的，而且这里不能使用整数数字。正确的语法是：

```
SELECT SUM(SALARY), EMP_ID  
FROM EMPLOYEE_PAY_TBL  
GROUP BY SALARY, EMP_ID;
```

b)

```
SELECT EMP_ID, MAX(SALARY)
FROM EMPLOYEE_PAY_TBL
GROUP BY SALARY, EMP_ID;
```

答：这个语句可以执行。

c)

```
SELECT EMP_ID, COUNT(SALARY)
FROM EMPLOYEE_PAY_TBL
ORDER BY EMP_ID
GROUP BY SALARY;
```

答：这个语句不能执行。ORDER BY子句和GROUP BY子句的次序不正确，另外，GROUP BY子句里必须有EMP\_ID字段。正确的语法是：

```
SELECT EMP_ID, COUNT(SALARY)
FROM EMPLOYEE_PAY_TBL
GROUP BY EMP_ID
ORDER BY EMP_ID;
```

d)

```
SELECT YEAR(Date_Hire) AS Year_Hired, SUM(SALARY)
FROM EMPLOYEE_PAY_TBL
GROUP BY 1
HAVING SUM(SALARY) > 20000;
```

答：这个语句可以执行。

2. 判断正误：在使用HAVING子句时一定也要使用GROUP BY子句。

答：错。使用HAVING子句不是一定要使用GROUP BY子句。

3. 判断正误：下面的SQL语句返回分组的薪水总和：

```
SELECT SUM(SALARY)
FROM EMPLOYEE_PAY_TBL;
```

答：错，这个语句里没有GROUP BY子句，所以不能返回分组的薪水总和。

4. 判断正误：被选中的字段在GROUP BY子句里必须以相同次序出现。

答：错。SELECT子句里的字段与GROUP BY子句里的字段可以具有不同次序。

5. 判断正误：HAVING子句告诉GROUP BY子句要包括哪些分组。

答：对。

练习答案

1. 不需要答案。
2. 不需要答案。
3. 不需要答案。
4. 修改练习3里的查询，把结果按降序排序，也就是数值从大到小。

答：

```
SELECT CITY, COUNT(*)FROM EMPLOYEE_TBL
GROUP BY CITY
ORDER BY 2 DESC;
```

5. 编写一个查询，从表EMPLOYEE\_PAY\_RATE里列出每个城市的平均税率和工资。

答：

```
SELECT POSITION, AVG(PAY_RATE)
FROM EMPLOYEE_PAY_TBL
GROUP BY POSITION;
```

6. 编写一个查询，从表EMPLOYEE\_PAY\_RATE里列出城市平均薪水高于\$20 000的每个城市的平均薪水。

答：

```
SELECT POSITION, AVG(SALARY)
FROM EMPLOYEE_PAY_TBL
GROUP BY POSITION
HAVING AVG(SALARY)>20000;
```

## 第11章

### 测验答案

1. 匹配函数与其描述。

答：

描述 函数

- a. 从字符串里选择一部分 SUBSTR
- b. 从字符串左侧或右侧剪切字符串 LTRIM/RTRIM
- c. 把全部字符都改变为大写 UPPER
- d. 确定字符串的长度 LENGTH
- e. 连接字符串 ||

2. 判断正误：在 SELECT 语句里使用函数调整数据输出外观时会影响数据库里存储的数据。

答：错。

3. 判断正误：当查询里出现函数嵌套时，最外层的函数会首先被处理。

答：错。最内层的函数会首先被处理。

## 练习答案

1. 不需要答案。

2. 不需要答案。

3. 编写一个SQL语句，列出雇员的电子邮件地址。电子邮件地址并不是数据库里的一个字段，雇员的电子邮件地址应该由以下形式构成：

`FIRST.LAST @PERPTECH.COM`

举例来说，John Smith的电子邮件地址是  
`JOHN.SMITH@PERPTECH.COM`。

答：

```
SELECT CONCAT(FIRST_NAME, '.', LAST_NAME, '@PERPTECH.COM')  
FROM EMPLOYEE_TBL;
```

4. 编写一个SQL语句，以如下形式列出雇员的姓名、ID和电话号码。

- a. 姓名显示为SMITH, JOHN;
- b. 雇员ID显示为999-99-9999;
- c. 电话号码显示为(999)999-9999。

答：

```
SELECT CONCAT(LAST_NAME, ', ', FIRST_NAME), EMP_ID,  
CONCAT('(', SUBSTRING(PHONE, 1, 3), ')', SUBSTRING(PHONE, 4, 3), '-',  
SUBSTRING(PHONE, 7, 4))  
FROM EMPLOYEE_TBL;
```

## 第12章

### 测验答案

1. 系统日期和时间源自于哪里？

答：系统日期和时间源自于主机操作系统的当前日期和时间。

2. 列出DATETIME值的标准内部元素？

答：YEAR、MONTH、DAY、HOUR、MINUTE和SECOND。

3. 如果公司是个国际公司，在处理日期和时间的比较与表示时，应该考虑的一个重要因素是什么？

答：时区。

4. 字符串表示的日期值能不能与定义为DATETIME类型的日期值进行比较？

答：DATETIME数据类型不能与定义字符串的日期值进行准确的比较，字符串必须首先转换为DATETIME数据类型。

5. 在SQL Server、MySQL和Oracle里，使用什么函数获取当前日期和时间？

答：NOW()。

练习答案

1. 不需要答案。

2. 不需要答案。

3. 不需要答案。

4. 不需要答案。

5. 不需要答案。

6. 每名雇员是在星期几被雇用的？

答：利用如下语句获得数据：

```
SELECT EMP_ID, DAYNAME(DATE_HIRE)
FROM EMPLOYEE_PAY_TBL;
```

7. 今天的儒略日期是多少（积日）？

答：使用如下语句获得数据：

```
SELECT DAYOFYEAR(CURRENT_DATE);
```

8. 不需要答案。

## 第13章

### 测验答案

1. 如果不论相关表里是否存在匹配的记录，都要从表里返回记录，应该使用什么类型的结合？

答：使用外向结合。

2. JOIN条件位于SQL语句的什么位置？

答：JOIN条件位于WHERE子句里。

3. 使用什么类型的结合来判断相关表的记录之间的相等关系？

答：相等结合。

4. 如果从两个不同的表获取数据，但它们没有结合，会产生什么结果？

答：我们会得到表的笛卡尔积（这也被称为交叉结合）。

5. 使用如下的表：

#### ORDERS\_TBL

ORD_NUM	VARCHAR2(10)	NOT NULL	primary key
CUST_ID	VARCHAR2(10)	NOT NULL	
PROD_ID	VARCHAR2(10)	NOT NULL	
QTY	INTEGER	NOT NULL	
ORD_DATE	DATE		

#### PRODUCTS\_TBL

PROD_ID	VARCHAR2(10)	NOT NULL	primary key
PROD_DESC	VARCHAR2(40)	NOT NULL	
COST	DECIMAL(,2)	NOT NULL	

下面使用外部结合的语法正确吗？

```
SELECT C.CUST_ID, C.CUST_NAME, O.ORD_NUM
FROM CUSTOMER_TBL C, ORDERS_TBL O
WHERE C.CUST_ID(+) = O.CUST_ID(+)
```

答：不正确。加号（+）应该只在WHERE子句里的O.CUST\_ID字段之后。正确的语法是：

```
SELECT C.CUST_ID, C.CUST_NAME, O.ORD_NUM
FROM CUSTOMER_TBL C, ORDERS_TBL O
WHERE C.CUST_ID = O.CUST_ID(+)
```

如果使用繁琐语法，上述查询语句会是什么样子？

```
SELECT C.CUST_ID, C.CUST_NAME, O.ORD_NUM
FROM CUSTOMER_TBL C LEFT OUTER JOIN ORDERS_TBL O
ON C.CUST_ID = O.CUST_ID
```

练习答案

1. 不需要答案。
2. 不需要答案。
3. 改写练习2里的SQL查询语句，使用 INNER JOIN语法。

答：

```
SELECT E.LAST_NAME, E.FIRST_NAME, EP.DATE_HIRE
FROM EMPLOYEE_TBL E INNER JOIN
EMPLOYEE_PAY_TBL EP ON
E.EMP_ID = EP.EMP_ID;
```

4. 编写一个SQL语句，从表EMPLOYEE\_TBL返回EMP\_ID、LAST\_NAME和FIRST\_NAME字段，从表EMPLOYEE\_PAY\_TBL返回SALARY和BONUS字段。使用两种类型的 INNER JOIN技术。完成上述查询以后，再进一步计算出每个城市雇员的平均薪水是多少。

答：



```
SELECT E.EMP_ID, E.LAST_NAME, E.FIRST_NAME, EP.SALARY, EP.BONUS
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL EP
WHERE E.EMP_ID = EP.EMP_ID;
```

```
SELECT E.EMP_ID, E.LAST_NAME, E.FIRST_NAME, EP.SALARY, EP.BONUS

FROM EMPLOYEE_TBL E INNER JOIN
EMPLOYEE_PAY_TBL EP
ON E.EMP_ID = EP.EMP_ID;
```

```
SELECT E.CITY, AVG(EP.SALARY) AVG_SALARY
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL EP
WHERE E.EMP_ID = EP.EMP_ID
GROUP BY E.CITY;
```

```
SELECT E.CITY, AVG(EP.SALARY) AVG_SALARY
FROM EMPLOYEE_TBL E INNER JOIN
EMPLOYEE_PAY_TBL EP
ON E.EMP_ID = EP.EMP_ID
GROUP BY E.CITY;
```

5. 不需要答案。

## 第14章

### 测验答案

1. 在用于SELECT语句时，子查询的功能是什么？

答：在用于SELECT语句时，子查询的主要功能是返回主查询需要的数据。

2. 在子查询与UPDATE语句配合使用时，能够更新多个字段吗？

答：是，使用一个UPDATE和子查询语句可以同时更新多个字段。

3. 下面的语法正确吗？如果不正确，那正确的语法应该是怎样？

a)

```

SELECT CUST_ID, CUST_NAME
  FROM CUSTOMER_TBL
 WHERE CUST_ID =
        (SELECT CUST_ID
          FROM ORDERS_TBL
         WHERE ORD_NUM = '16C17');

```

答：语法正确。

b)

```

SELECT EMP_ID, SALARY
  FROM EMPLOYEE_PAY_TBL
 WHERE SALARY BETWEEN '20000'
        AND (SELECT SALARY
              FROM EMPLOYEE_PAY_TBL
             WHERE SALARY = '40000');

```

答：不正确。操作符BETWEEN不能以这种格式使用。

c)

```

UPDATE PRODUCTS_TBL
  SET COST = 1.15
 WHERE PROD_ID =
        (SELECT PROD_ID
          FROM ORDERS_TBL
         WHERE ORD_NUM = '32A132');

```

答：语法正确。

4. 下面语句执行的结果是什么？

```

DELETE FROM EMPLOYEE_TBL
 WHERE EMP_ID IN
        (SELECT EMP_ID
          FROM EMPLOYEE_PAY_TBL);

```

答：表EMPLOYEE\_TBL里与表EMPLOYEE\_PAY\_TBL里具有相同

EMP\_ID的记录会被全部删除。这时，我们强烈推荐在子查询里使用WHERE子句。

练习答案

1. 不需要答案。
2. 使用子查询编写一个SQL语句来更新表CUSTOMER\_TBL，找到ORD\_NUM列中订单号码为 23E934的顾客，把顾客名称修改为DAVIDS MARKET。

答：

```
UPDATE CUSTOMER_TBL
  SET CUST_NAME = 'DAVIDS MARKET'
  WHERE CUST_ID =
      (SELECT CUST_ID
       FROM ORDERS_TBL
       WHERE ORD_NUM = '23E934');
```

3. 使用子查询编写一个SQL语句，返回工资高于 JOHN DOE的全部雇员的姓名；JOHN DOE的雇员标识号码是 343559876。

答：

```
SELECT E.LAST_NAME, E.FIRST_NAME, E.MIDDLE_NAME
  FROM EMPLOYEE_TBL E,
       EMPLOYEE_PAY_TBL P
 WHERE P.PAY_RATE > (SELECT PAY_RATE
                     FROM EMPLOYEE_PAY_TBL
                     WHERE EMP_ID = '343559876');
```

4. 使用子查询编写一个SQL语句，列出所有价格高于全部产品平均价格的产品。

答：

```

SELECT PROD_DESC
FROM PRODUCTS_TBL
WHERE COST > (SELECT AVG(COST)
              FROM PRODUCTS_TBL);

```

## 第15章

### 测验答案

在下面的练习里使用INTERSECT或EXCEPT操作符时，请参考Oracle的语法。

1. 下面组合查询的语法正确吗？如果不正确，请修改它们。它们使用的表EMPLOYEE\_TBL和EMPLOYEE\_PAY\_TBL如下所示：

#### EMPLOYEE\_TBL

EMP_ID	VARCHAR(9)	NOT NULL,
LAST_NAME	VARCHAR(15)	NOT NULL,
FIRST_NAME	VARCHAR(15)	NOT NULL,
MIDDLE_NAME	VARCHAR(15),	
ADDRESS	VARCHAR(30)	NOT NULL,
CITY	VARCHAR(15)	NOT NULL,
STATE	VARCHAR(2)	NOT NULL,
ZIP	INTEGER(5)	NOT NULL,
PHONE	VARCHAR(10),	
PAGER	VARCHAR(10),	

#### EMPLOYEE\_PAY\_TBL

EMP_ID	VARCHAR(9)	NOT NULL,	primary key
POSITION	VARCHAR(15)	NOT NULL,	
DATE_HIRE	DATETIME,		
PAY_RATE	DECIMAL(4,2)	NOT NULL,	
DATE_LASTRAISE	DATE,		
SALARY	DECIMAL(8,2),		
BONUS	DECIMAL(6,2),		

a)

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME
FROM EMPLOYEE_TBL
UNION
SELECT EMP_ID, POSITION, DATE_HIRE
FROM EMPLOYEE_PAY_TBL;
```

答：这个组合查询不会执行，因为数据类型不匹配。EMP\_ID字段是匹配的，但LAST\_NAME和FIRST\_NAME并不匹配POSITION和DATE\_HIRE数据类型。

b)

```
SELECT EMP_ID FROM EMPLOYEE_TBL
UNION ALL
SELECT EMP_ID FROM EMPLOYEE_PAY_TBL
ORDER BY EMP_ID;
```

答：语句正确。

c)

```
SELECT EMP_ID FROM EMPLOYEE_PAY_TBL
INTERSECT
SELECT EMP_ID FROM EMPLOYEE_TBL
ORDER BY 1;
```

答：这个组合查询会正常执行。

2. 匹配操作符与相应的描述。

描述	操作符
a. 显示重复记录	UNION ALL
b. 返回第一个查询里与第二个查询匹配的结果	INTERSECT
c. 返回不重复的记录	UNION
d. 返回第一个查询里有但第二个查询没有的结果	EXCEPT

## 练习答案

下面的练习请参考本章介绍的语法。由于 MySQL 不支持本章介绍的两个操作符，所以请自行编写查询语句，并与书中提供的进行比较。

使用的表CUSTOMER\_TBL和ORDERS\_TBL如下所示：

CUSTOMER_TBL			
CUST_IN	VARCHAR(10)	NOT NULL	primary key
CUST_NAME	VARCHAR(30)	NOT NULL,	
CUST_ADDRESS	VARCHAR(20)	NOT NULL,	
CUST_CITY	VARCHAR(15)	NOT NULL,	
CUST_STATE	VARCHAR(2)	NOT NULL,	
CUST_ZIP	INTEGER(5)	NOT NULL,	
CUST_PHONE	INTEGER(10),		
CUST_FAX	INTEGER(10)		

ORDERS_TBL			
ORD_NUM	VARCHAR(10)	NOT NULL	primary key
CUST_ID	VARCHAR(10)	NOT NULL,	
PROD_ID	VARCHAR(10)	NOT NULL,	
QTY	INTEGER(6)	NOT NULL,	
ORD_DATE	DATETIME		

3. 编写一个组合查询，返回下了订单的顾客。

答：

```
SELECT CUST_ID FROM CUSTOMER_TBL
INTERSECT
SELECT CUST_ID FROM ORDERS_TBL;
```

4. 编写一个组合查询，返回没有下订单的顾客。

答：

```
SELECT CUST_ID FROM CUSTOMER_TBL
EXCEPT
SELECT CUST_ID FROM ORDERS_TBL;
```

## 第16章

### 测验答案

1. 使用索引的主要缺点是什么？

答：索引的主要缺点包括会减缓批处理操作、占据磁盘空间、维护开销。

2. 组合索引里的字段顺序为什么很重要？

答：因为先执行最严格的条件就会改善性能。

3. 具有大量NULL值的字段是否应该设置索引？

答：不，具有大量NULL值的字段不应该设置索引。当很多记录的值相同时，访问它们的速度会因此而下降。

4. 索引的主要作用是去除表里的重复数据吗？

答：不是。索引的主要作用是提高数据检索速度。当然，唯一索引会禁止表里包含重复数据。

5. 判断正误：使用组合索引的主要原因是在索引里使用汇聚函数。

答：错。组合索引的主要是对同一个表里的两个或多个字段设置索引。

6. 基数是什么含义？什么样的字段可以被看作是高基数的？

答：基数是指数据在字段里的唯一性。SSN（社会保险号码）就是这种字段的一个范例。

### 练习答案

1. 判断在下列情况下是否应该使用索引，如果是，请选择索引的类型。

A. 字段很多，但表的规模相对较小。

答：小规模表不需要设置索引。

B. 中等规模的表，不允许有重复值。

答：可以使用唯一索引。

C. 多个字段，大规模的表，多个字段用在WHERE子句作为过滤器。

答：可以针对WHERE子句里使用的字段设置组合索引。

D. 大规模表，很多字段，大量数据操作。

答：可以根据过滤、排序和分级的要求设置单字段索引或组合索引。对于大规模数据操作来说，可以在执行INSERT、UPDATE或DELETE语句之前删除索引，之后再重新创建。

2. 不需要答案。

3. 修改练习2所创建的索引，将其变成唯一索引。要为SALARY字段创建唯一索引，需要做些什么？编写并依次运行这些命令。

答：

```
DROP INDEX EP_POSITON ON EMPLOYEE_PAY_TBL;  
CREATE UNIQUE INDEX EP_POSITION  
ON EMPLOYEE_TBL (POSITION);
```

4. 研究本书里使用的表，根据用户可能对表进行的检索方式，判断哪些字段适合设置索引。

答：

```
EMPLOYEE_TBL.LAST_NAME  
EMPLOYEE_TBL.FIRST_NAME  
EMPLOYEE_TBL.EMP_ID  
EMPLOYEE_PAY_TBL.EMP_ID  
EMPLOYEE_PAY_TBL.POSITION  
CUSTOMER_TBL.CUST_ID  
CUSTOMER_TBL.CUST_NAME  
ORDERS_TBL.ORD_NUM  
ORDERS_TBL.CUST_ID  
ORDERS_TBL.PROD_ID  
ORDERS_TBL.ORD_DATE  
PRODUCTS_TBL.PROD_ID  
PRODUCTS_TBL.PROD_DESC
```



5. 在表 ORDERS\_TBL 上创建一个多字段索引，包含下列字段：  
CUST\_ID、PROD\_ID和ORD\_DATE。

答：

```
CREATE INDEX ORD_IDX ON ORDERS_TBL (CUST_ID, PROD_ID, ORD_DATE);
```

6. 答案不确定。

## 第17章

### 测验答案

1. 在小规模表上使用唯一索引会带来什么好处吗？

答：这个索引对于性能来说没有任何好处，但有助于保持引用完整性。关于引用完整性请见第3章。

2. 当查询执行时，如果优化器决定不使用表上的索引，会发生什么呢？

答：全表扫描。

3. WHERE子句里的最严格条件应该放在结合条件之前还是之后呢？

答：最严格条件应该在结合条件之前求值，因此结合条件通常会返回大量的数据。

### 练习答案

1. 改写下面的 SQL 语句来改善性能。使用如下所示的表  
EMPLOYEE\_TBL 和表EMPLOYEE\_PAY\_TBL。

```

EMPLOYEE_TBL
EMP_ID      VARCHAR(9)      NOT NULL      Primary key
LAST_NAME   VARCHAR(15)   NOT NULL,
FIRST_NAME  VARCHAR(15)   NOT NULL,
MIDDLE_NAME VARCHAR(15),
ADDRESS     VARCHAR(30)   NOT NULL,
CITY        VARCHAR(15)   NOT NULL,
STATE       VARCHAR(2)    NOT NULL,
ZIP         INTEGER(5)     NOT NULL,
PHONE       VARCHAR(10),
PAGER       VARCHAR(10),
EMPLOYEE_PAY_TBL
EMP_ID      VARCHAR(9)      NOT NULL      primary key
POSITION    VARCHAR(15)     NOT NULL,
DATE_HIRE   DATETIME,
PAY_RATE    DECIMAL(4,2)    NOT NULL,
DATE_LAST_RAISE DATETIME,
SALARY      DECIMAL(8,2),
BONUS       DECIMAL(8,2),

```

a)

```

SELECT EMP_ID, LAST_NAME, FIRST_NAME,
       PHONE
FROM EMPLOYEE_TBL

WHERE SUBSTRING(PHONE, 1, 3) = '317' OR
      SUBSTRING(PHONE, 1, 3) = '812' OR
      SUBSTRING(PHONE, 1, 3) = '765';

```

答:

```

SELECT EMP_ID, LAST_NAME, FIRST_NAME,
       PHONE
FROM EMPLOYEE_TBL
WHERE SUBSTRING(PHONE, 1, 3) IN ('317', '812', '765');

```

根据经验，把OR条件转换为IN列表会更好一些。

b)

```
SELECT LAST_NAME, FIRST_NAME  
FROM EMPLOYEE_TBL  
WHERE LAST_NAME LIKE '%ALL%';
```

答：

```
SELECT LAST_NAME, FIRST_NAME  
FROM EMPLOYEE_TBL  
WHERE LAST_NAME LIKE 'WAL%';
```

如果在条件值里不包含首字符，就不能发挥索引的作用。

c)

```
SELECT E.EMP_ID, E.LAST_NAME, E.FIRST_NAME,  
EP.SALARY  
FROM EMPLOYEE_TBL E,  
EMPLOYEE_PAY_TBL EP  
WHERE LAST_NAME LIKE 'S%'  
AND E.EMP_ID = EP.EMP_ID;
```

答：

```
SELECT E.EMP_ID, E.LAST_NAME, E.FIRST_NAME,  
EP.SALARY  
FROM EMPLOYEE_TBL E,  
EMPLOYEE_PAY_TBL EP  
WHERE E.EMP_ID = EP.EMP_ID  
AND LAST_NAME LIKE 'S%';
```

2. 添加一个名为EMPLOYEE\_PAYHIST\_TBL的表，用于存放大量的支付历史数据。使用下面的表来编写SQL语句，解决后续的问题。

```

EMPLOYEE_PAYHIST_TBL
PAYHIST_ID      VARCHAR(9)      NOT NULL      primary key,
EMP_ID          VARCHAR(9)      NOT NULL,
START_DATE      DATETIME      NOT NULL,
END_DATE        DATETIME,
PAY_RATE        DECIMAL(4,2)   NOT NULL,
SALARY          DECIMAL(8,2)   NOT NULL,
BONUS          DECIMAL(8,2)   NOT NULL,
CONSTRAINT EMP_FK FOREIGN KEY (EMP_ID)
REFERENCES EMPLOYEE_TBL (EMP_ID)

```

首先思考，用什么方法能够确定所写的查询可以正确执行？

a. 查询正式员工（salaried employee）和非正式员工（nonsalaried employee）在付薪第一年各自的总人数。

答：

```

SELECT START_YEAR,SUM(SALARIED) AS SALARIED,SUM(HOURLY) AS
HOURLY
FROM
(SELECT YEAR(E.START_DATE) AS START_YEAR,COUNT(E.EMP_ID)
AS SALARIED,0 AS HOURLY
FROM EMPLOYEE_PAYHIST_TBL E INNER JOIN
( SELECT MIN(START_DATE) START_DATE,EMP_ID
FROM EMPLOYEE_PAYHIST_TBL
GROUP BY EMP_ID) F ON E.EMP_ID=F.EMP_ID AND
E.START_DATE=F.START_DATE
WHERE E.SALARY > 0.00
GROUP BY YEAR(E.START_DATE)
UNION
SELECT YEAR(E.START_DATE) AS START_YEAR,0 AS SALARIED,
COUNT(E.EMP_ID) AS HOURLY
FROM EMPLOYEE_PAYHIST_TBL E INNER JOIN
( SELECT MIN(START_DATE) START_DATE,EMP_ID
FROM EMPLOYEE_PAYHIST_TBL
GROUP BY EMP_ID) F ON E.EMP_ID=F.EMP_ID AND
E.START_DATE=F.START_DATE
WHERE E.PAY_RATE > 0.00
GROUP BY YEAR(E.START_DATE)
) A
GROUP BY START_YEAR
ORDER BY START_YEAR

```

b. 查询正式员工和非正式员工在付薪第一年各自总人数的差异。  
其中，非正式员工全年无缺勤（ $PAY\_RATE * 52 * 40$ ）。

答：

```

SELECT START_YEAR,SALARIED AS SALARIED,HOURLY AS HOURLY,
      (SALARIED - HOURLY) AS PAY_DIFFERENCE
FROM
      (SELECT YEAR(E.START_DATE) AS START_YEAR,AVG(E.SALARY) AS
SALARIED,
      0 AS HOURLY
FROM EMPLOYEE_PAYHIST_TBL E INNER JOIN
      ( SELECT MIN(START_DATE) START_DATE,EMP_ID
FROM EMPLOYEE_PAYHIST_TBL
GROUP BY EMP_ID) F ON E.EMP_ID=F.EMP_ID AND
E.START_DATE=F.START_DATE
WHERE E.SALARY > 0.00
GROUP BY YEAR(E.START_DATE)
UNION
SELECT YEAR(E.START_DATE) AS START_YEAR,0 AS SALARIED,
      AVG(E.PAY_RATE * 52 * 40 ) AS HOURLY
FROM EMPLOYEE_PAYHIST_TBL E INNER JOIN
      ( SELECT MIN(START_DATE) START_DATE,EMP_ID
FROM EMPLOYEE_PAYHIST_TBL
GROUP BY EMP_ID) F ON E.EMP_ID=F.EMP_ID AND
E.START_DATE=F.START_DATE
WHERE E.PAY_RATE > 0.00
GROUP BY YEAR(E.START_DATE)
) A
GROUP BY START_YEAR
ORDER BY START_YEAR

```

c. 查询正式员工现在和刚入职时的薪酬差别。同样，非正式员工全年无缺勤。并且，员工的薪水在EMPLOYEE\_PAY\_TBL和EMPLOYEE\_PAYHIST\_TBL两个表中都有记录。在支付历史表中，当前支付记录的END\_DATE字段为NULL值。

答：

```

SELECT CURRENTPAY.EMP_ID,STARTING_ANNUAL_PAY,CURRENT_
ANNUAL_PAY,
CURRENT_ANNUAL_PAY - STARTING_ANNUAL_PAY AS PAY_DIFFERENCE
FROM
(SELECT EMP_ID,(SALARY + (PAY_RATE * 52 * 40)) AS
CURRENT_ANNUAL_PAY
FROM EMPLOYEE_PAYHIST_TBL
WHERE END_DATE IS NULL) CURRENTPAY
INNER JOIN
(SELECT E.EMP_ID,(SALARY + (PAY_RATE * 52 * 40)) AS
STARTING_ANNUAL_PAY
FROM EMPLOYEE_PAYHIST_TBL E
( SELECT MIN(START_DATE) START_DATE,EMP_ID
FROM EMPLOYEE_PAYHIST_TBL
GROUP BY EMP_ID) F ON E.EMP_ID=F.EMP_ID AND
E.START_DATE=F.START_DATE
) STARTINGPAY ON
CURRENTPAY.EMP_ID = STARTINGPAY.EMP_ID

```

## 第18章

### 测验答案

1. 使用什么命令创建会话？

答：CONNECT TO语句。

2. 在删除包含数据库对象的规划时，必须要使用什么选项？

答：使用CASCADE选项可以删除包含对象的规划。

3. MySQL里使用什么命令创建规划？

答：CREATE SCHEMA命令。

4. 使用什么命令撤销数据库权限？

答：REVOKE命令用于撤销数据库权限。

5. 什么命令能够创建表、视图和权限的组或集合？

答：CREATE SCHEMA语句。

6. 在SQL Server中，登录账户和数据库账户有什么区别？

答：登录账户可以登录SQL Server实例并访问资源。数据库账户可

以访问数据库并被赋予了相应的权限。

### 练习答案

1. 描述如何在learnsql数据库里创建一个新用户“John”。

答：

```
USE LEARNSQL;  
CREATE USER JOHN
```

2. 如何让新用户John能够访问表Employee\_tbl。

答： `GRANT SELECT ON TABLE EMPLOYEE_TBI TO JOHN;`

3. 描述如何设置John的权限，让它访问learnsql数据库里的全部对象？

答： `GRANT SELECT ON TABLE * TO JOHN;`

4. 描述如何撤销John的权限并且删除他的账户。

答： `DROP USER JOHN CASCADE;`

## 第19章

### 测验答案

1. 如果用户要把不是其所属对象的权限授予另一个用户，必须具有什么选项？

答： `GRANT OPTION`

2. 当权限被授予PUBLIC之后，是数据库的全部用户，还是仅特定用户获得这些权限？

答：数据库的全部用户都会获得这些权限。

3. 查看指定表里的数据需要什么权限？

答：SELECT权限。

4. SELECT是什么类型的权限？

答：对象级权限。



5. 如果想撤销用户对某个对象的权限，以及其他使用GRANT分配了这个对象权限的用户的权限，应该使用什么选项？

答：在REMOVE命令里使用CASCADE选项可以删除该对象分配出去的权限。

练习答案

1. 不需要答案。
2. 不需要答案。
3. 不需要答案。
4. 不需要答案。

第20章

测验答案

1. 在一个基于多个表创建的视图里，我们可以删除记录吗？

答：不能。只有基于单个表创建的视图才能使用DELETE、INSERT和UPDATE命令。

2. 在创建一个表时，所有者会自动被授予适当的权限。在创建视图时也是这样吗？

答：是的。视图所有者自动被授予关于视图的适当权限。

3. 在创建视图时，使用什么子句对数据进行排序？

答：在视图里GROUP BY子句起到了普通查询中ORDER BY子句（或GROUP BY子句）的作用。

4. 在基于视图创建视图时，使用什么选项检查完整性约束？

答：WITH CHECK OPTION。

5. 在尝试删除视图时，由于存在着多个底层视图，操作出现了错误。这时怎样做才能删除视图？

答：在DROP语句里添加CASCADE选项，这样可以删除全部的底层视图。

练习答案

1. 编写一个语句，基于表EMPLOYEE\_TBL的全部内容创建一个视图。

答：

```
CREATE VIEW EMP_VIEW AS  
SELECT * FROM EMPLOYEE_TBL;
```

2. 编写一个语句创建一个包含摘要数据的视图，显示表EMPLOYEE\_TBL 里每个城市的平均章工资和平均薪水。

答：

```
CREATE VIEW AVG_PAY_VIEW AS  
SELECT E.CITY, AVG(P.PAY_RATE), AVG(P.SALARY)  
FROM EMPLOYEE_PAY_TBL P,  
EMPLOYEE_TBL E  
WHERE P.EMP_ID = E.EMP_ID  
GROUP BY E.CITY;
```

3. 再次创建练习2中的摘要数据视图，但不要使用表EMPLOYEE\_TBL，而是使用练习1中所创建的视图。比较两个结果。

答：

```
CREATE VIEW AVG_PAY_ALT_VIEW AS  
SELECT E.CITY, AVG(P.PAY_RATE), AVG(P.SALARY)  
FROM EMPLOYEE_PAY_TBL P,  
EMP_VIEW E  
WHERE P.EMP_ID = E.EMP_ID  
GROUP BY E.CITY;
```

4. 使用练习2中创建的视图来创建一个名为EMPLOYEE\_PAY\_SUMMARIZED的表。想办法确定视图和表拥有相同的数据。

答：

```
SELECT * INTO EMPLOYEE_PAY_SUMMARIZED FROM AVG_PAY_VIEW;
```

5. 编写SQL语句来删除表和刚刚创建的视图。

答：

```
DROP VIEW EMP_VIEW;  
DROP VIEW AVG_PAY_VIEW;  
DROP VIEW AVG_PAY_ALT_VIEW;
```

## 第21章

### 测验答案

1. 在某些实现里，系统目录也被称为什么？

答：系统目录也被称为“数据目录”。

2. 普通用户能够更新系统目录吗？

答：不能直接更新。但是当用户创建对象时，系统目录会自动更新。

3. 在Microsoft SQL Server里哪个系统表格包含了数据库里视图的信息？

答：SYSVIEWS。

4. 谁拥有系统目录？

答：系统目录的拥有者通常是名为SYS或SYSTEM的用户，它也可以属于数据库的其他用户，但通常不会属于数据库里某个特定规划。

5. Oracle数据对象ALL\_TABLES和DBA\_TABLES之间的区别是什么？

答：ALL\_TABLES包含由特定用户访问的全部表，而DBA\_TABLES包含数据库里的全部表。

6. 谁修改系统表格？

答：数据库服务程序。

### 练习答案

1. 不需要答案。
2. 不需要答案。
3. 不需要答案。

## 第22章

### 测验答案

1. 触发器能够被修改吗？

答：触发器必须被替换或重新创建。

2. 当光标被关闭之后，我们能够重用它的名称吗？

答：这取决于具体的实现。在某些实现里，关闭光标之后就可以重新使用它的名称、释放内存，而其他一些实现必须先使用 DEALLOCATE 语句，然后才能重用它的名称。

3. 当光标被打开之后，使用什么命令获取它的结果？

答：FETCH 命令。

4. 触发器能够在 INSERT、DELETE 或 UPDATE 语句之前或之后执行吗？

答：触发器能够在 INSERT、DELETE 或 UPDATE 语句之前或之后执行，而且触发器有多种类型。

5. 在 MySQL 里使用什么语句从 XML 片断里获取信息？

答：EXTRACTVALUE 语句。

6. 为什么 Oracle 和 MySQL 不支持针对光标的 DEALLOCATE 语法？

答：因为在光标被关闭之后，这两种 SQL 实现会自动释放光标的资源。

7. 为什么光标不是基于数据集的操作？

答：光标不是基于数据集的操作，是因为光标每次只作用于一行数据，它将数据从内存中取出并进行相应的操作。

### 练习答案

1. 不需要答案。

2. 编写一个 SELECT 语句来生成 SQL 代码，统计每个表里的记录数量。（提示：类似于练习1。）

答：

```
SELECT CONCAT('SELECT COUNT(*) FROM ',TABLE_NAME,';') FROM
TABLES;
```

3. 编写一组SQL命令来创建一个光标，返回所有用户及其销售数据。确保在用户所使用的实现中，正确关闭光标并回收资源。

答：

An example using SQL Server might look similar to this:

```
BEGIN
    DECLARE @custname VARCHAR(30);
    DECLARE @purchases decimal(6,2);
    DECLARE customercursor CURSOR FOR SELECT
    C.CUST_NAME,SUM(P.COST*O.QTY) as SALES
    FROM CUSTOMER_TBL C
    INNER JOIN ORDERS_TBL O ON C.CUST_ID=O.CUST_ID
    INNER JOIN PRODUCTS_TBL P ON O.PROD_ID=P.PROD_ID
    GROUP BY C.CUST_NAME;
    OPEN customercursor;
    FETCH NEXT FROM customercursor INTO @custname,@purchases
    WHILE (@@FETCH_STATUS<>-1)
        BEGIN
            IF (@@FETCH_STATUS<>-2)
                BEGIN
                    PRINT @custname + ': $' + CAST(@purchases AS
VARCHAR(20))
                END
            FETCH NEXT FROM customercursor INTO @custname,@purchases
        END
    CLOSE customercursor
    DEALLOCATE customercursor
END;
```

## 测验答案

1. 一台服务器上的数据库能够被另一台服务器访问吗？

答：可以，通过使用中间件，这被称为访问远程数据库。

2. 公司可以使用什么方式向自己的雇员发布信息？

答：内部网。

3. 提供对数据库连接的产品被称为什么？

答：中间件。

4. SQL能够嵌入到互联网编程语言里吗？

答：可以。SQL可以嵌入到互联网编程语言，比如Java。

5. 如何通过Web程序访问远程数据库？

答：通过Web服务器。

## 练习答案

1. 答案不确定。

2. 不需要答案。

## 第24章

## 测验答案

1. SQL是过程语言还是非过程语言？

答：SQL 是非过程语言，表示数据库决定如何执行 SQL 语句。本章介绍的扩展是过程语言。

2. 除了声明光标之外，光标的3个基本操作是什么？

答：OPEN、FETCH和CLOSE。

3. 过程或非过程：数据库发动机在处理什么语句时会决定对SQL语句进行估值和执行？

答：非过程语句。

## 练习答案

不需要答案。

## [附录D 本书范例的CREATE TABLE语句](#)

这个附录很有用，其中不仅列出了本书范例所使用的CREATE TABLE语句，还展示了不同数据库平台的语法差别。读者可以用这些语句创建自己的表格，从而完成书中的练习。

### [D.1 MySQL](#)

#### EMPLOYEE\_TBL

```
CREATE TABLE EMPLOYEE_TBL
(
  EMP_ID          VARCHAR(9)          NOT NULL,
  LAST_NAME       VARCHAR(15)         NOT NULL,
  FIRST_NAME      VARCHAR(15)         NOT NULL,
  MIDDLE_NAME     VARCHAR(15),
  ADDRESS         VARCHAR(30)         NOT NULL,
  CITY            VARCHAR(15)         NOT NULL,
  STATE           CHAR(2)              NOT NULL,
  ZIP             INTEGER(5)           NOT NULL,
  PHONE           CHAR(10),
  PAGER           CHAR(10),
  CONSTRAINT EMP_PK PRIMARY KEY (EMP_ID)
);
```

#### EMPLOYEE\_PAY\_TBL

```
CREATE TABLE EMPLOYEE_PAY_TBL
(
  EMP_ID          VARCHAR(9)          NOT NULL    primary key,
  POSITION         VARCHAR(15)         NOT NULL,
  DATE_HIRE       DATE,
  PAY_RATE        DECIMAL(4,2),
  DATE_LAST_RAISE DATE,
  SALARY          DECIMAL(8,2),
  BONUS           DECIMAL(6,2),
  CONSTRAINT EMP_FK FOREIGN KEY (EMP_ID) REFERENCES EMPLOYEE_TBL (EMP_ID)
);
```

#### CUSTOMER\_TBL

```

CREATE TABLE CUSTOMER_TBL
(
CUST_ID          VARCHAR(10)    NOT NULL          primary key,
CUST_NAME        VARCHAR(30)    NOT NULL,
CUST_ADDRESS     VARCHAR(20)    NOT NULL,
CUST_CITY        VARCHAR(15)    NOT NULL,
CUST_STATE       CHAR(2)        NOT NULL,
CUST_ZIP         INTEGER(5)     NOT NULL,
CUST_PHONE       CHAR(10),
CUST_FAX         INTEGER(10)
);

```

## ORDERS\_TBL

```

CREATE TABLE ORDERS_TBL
(
ORD_NUM          VARCHAR(10)    NOT NULL          primary key,
CUST_ID          VARCHAR(10)    NOT NULL,
PROD_ID          VARCHAR(10)    NOT NULL,
QTY              INTEGER(6)     NOT NULL,
ORD_DATE         DATE
);

```

## PRODUCTS\_TBL

```

CREATE TABLE PRODUCTS_TBL
(
PROD_ID          VARCHAR(10)    NOT NULL          primary key,
PROD_DESC        VARCHAR(40)    NOT NULL,
COST             DECIMAL(6,2)   NOT NULL
);

```

## [D.2 Oracle和SQL Server](#)

## EMPLOYEE\_TBL



```

CREATE TABLE EMPLOYEE_TBL
(
EMP_ID          VARCHAR(9)          NOT NULL,
LAST_NAME       VARCHAR(15)         NOT NULL,
FIRST_NAME      VARCHAR(15)         NOT NULL,
MIDDLE_NAME     VARCHAR(15),
ADDRESS         VARCHAR(30)         NOT NULL,
CITY            VARCHAR(15)         NOT NULL,
STATE          CHAR(2)              NOT NULL,
ZIP            INTEGER              NOT NULL,
PHONE          CHAR(10),
PAGER          CHAR(10),
CONSTRAINT EMP_PK PRIMARY KEY (EMP_ID)
);

```

## EMPLOYEE\_PAY\_TBL

```

CREATE TABLE EMPLOYEE_PAY_TBL
(
EMP_ID          VARCHAR(9)          NOT NULL    primary key,
POSITION        VARCHAR(15)         NOT NULL,
DATE_HIRE       DATE,
PAY_RATE        DECIMAL(4,2),
DATE_LAST_RAISE DATE,
SALARY          DECIMAL(8,2),
BONUS          DECIMAL(6,2),
CONSTRAINT EMP_FK FOREIGN KEY (EMP_ID) REFERENCES EMPLOYEE_TBL (EMP_ID)
);

```

## CUSTOMER\_TBL

```

CREATE TABLE CUSTOMER_TBL
(
CUST_ID         VARCHAR(10)         NOT NULL    primary key,
CUST_NAME       VARCHAR(30)         NOT NULL,
CUST_ADDRESS    VARCHAR(20)         NOT NULL,
CUST_CITY       VARCHAR(15)         NOT NULL,
CUST_STATE      CHAR(2)              NOT NULL,
CUST_ZIP        INTEGER              NOT NULL,
CUST_PHONE      CHAR(10),
CUST_FAX        VARCHAR(10)
);

```

## ORDERS\_TBL

```
CREATE TABLE ORDERS_TBL
(
ORD_NUM          VARCHAR(10)    NOT NULL    primary key,
CUST_ID          VARCHAR(10)    NOT NULL,
PROD_ID          VARCHAR(10)    NOT NULL,
QTY              INTEGER        NOT NULL,
ORD_DATE         DATE
);
```

## PRODUCTS\_TBL

```
CREATE TABLE PRODUCTS_TBL
(
PROD_ID          VARCHAR(10)    NOT NULL    primary key,
PROD_DESC        VARCHAR(40)    NOT NULL,
COST             DECIMAL(6,2)   NOT NULL
);
```

### [附录E 书中范例所涉数据的INSERT语句](#)

这个附录列出的INSERT语句用于填充附录D里的表格。创建了上述表格之后，我们就可以使用这些INSERT语句来填充数据。

#### [E.1 MySQL和SQL Server](#)

##### [E.1.1 EMPLOYEE\\_TBL](#)

```
INSERT INTO EMPLOYEE_TBL VALUES  
( '311549902', 'STEPHENS', 'TINA', 'DAWN', 'RR 3 BOX 17A', 'GREENWOOD',  
'IN', '47890', '3178784465', NULL);
```

```
INSERT INTO EMPLOYEE_TBL VALUES  
( '442346889', 'PLEW', 'LINDA', 'CAROL', '3301 BEACON', 'INDIANAPOLIS',  
'IN', '46224', '3172978990', NULL);
```

```
INSERT INTO EMPLOYEE_TBL VALUES  
( '213764555', 'GLASS', 'BRANDON', 'SCOTT', '1710 MAIN ST', 'WHITELAND',  
'IN', '47885', '3178984321', '3175709980');
```

```
INSERT INTO EMPLOYEE_TBL VALUES  
( '313782439', 'GLASS', 'JACOB', NULL, '3789 WHITE RIVER BLVD',  
'INDIANAPOLIS', 'IN', '45734', '3175457676', '8887345678');
```

```
INSERT INTO EMPLOYEE_TBL VALUES  
( '220984332', 'WALLACE', 'MARIAH', NULL, '7889 KEYSTONE AVE',  
'INDIANAPOLIS', 'IN', '46741', '3173325986', NULL);
```

```
INSERT INTO EMPLOYEE_TBL VALUES  
( '443679012', 'SPURGEON', 'TIFFANY', NULL, '5 GEORGE COURT',  
'INDIANAPOLIS', 'IN', '46234', '3175679007', NULL);
```

### E.1.2 EMPLOYEE\_PAY\_TBL

```
INSERT INTO EMPLOYEE_PAY_TBL VALUES  
( '311549902', 'MARKETING', '1999-05-23', NULL, '2009-05-01', '40000', NULL);
```

```
INSERT INTO EMPLOYEE_PAY_TBL VALUES  
( '442346889', 'TEAM LEADER', '2000-06-17', '14.75', '2009-06-01', NULL,  
NULL);
```

```
INSERT INTO EMPLOYEE_PAY_TBL VALUES  
( '213764555', 'SALES MANAGER', '2004-08-14', NULL, '2009-08-01', '30000',  
'2000');
```

```
INSERT INTO EMPLOYEE_PAY_TBL VALUES  
( '313782439', 'SALESMAN', '2007-06-28', NULL, NULL, '20000', '1000');
```

```
INSERT INTO EMPLOYEE_PAY_TBL VALUES  
( '220984332', 'SHIPPER', '2006-07-22', '11.00', '1999-07-01', NULL, NULL);
```

```
INSERT INTO EMPLOYEE_PAY_TBL VALUES  
( '443679012', 'SHIPPER', '2001-01-14', '15.00', '1999-01-01', NULL, NULL);
```

### E.1.3 CUSTOMER\_TBL

```
INSERT INTO CUSTOMER_TBL VALUES  
( '232', 'LESLIE GLEASON', '798 HARDAWAY DR', 'INDIANAPOLIS',  
'IN', '47856', '3175457690', NULL);
```

```
INSERT INTO CUSTOMER_TBL VALUES  
( '109', 'NANCY BUNKER', 'APT A 4556 WATERWAY', 'BROAD RIPPLE',  
'IN', '47950', '3174262323', NULL);
```

```
INSERT INTO CUSTOMER_TBL VALUES  
( '345', 'ANGELA DOBK0', 'RR3 BOX 76', 'LEBANON', 'IN', '49967',  
'7658970090', NULL);
```

```
INSERT INTO CUSTOMER_TBL VALUES  
( '090', 'WENDY WOLF', '3345 GATEWAY DR', 'INDIANAPOLIS', 'IN',  
'46224', '3172913421', NULL);
```

```
INSERT INTO CUSTOMER_TBL VALUES  
( '12', 'MARYS GIFT SHOP', '435 MAIN ST', 'DANVILLE', 'IL', '47978',  
'3178567221', '3178523434');
```

```
INSERT INTO CUSTOMER_TBL VALUES  
( '432', 'SCOTTYS MARKET', 'RR2 BOX 173', 'BROWNSBURG', 'IN',  
'45687', '3178529835', '3178529836');
```

```
INSERT INTO CUSTOMER_TBL VALUES  
( '333', 'JASONS AND DALLAS GOODIES', 'LAFAYETTE SQ MALL',  
'INDIANAPOLIS', 'IN', '46222', '3172978886', '3172978887');
```

```
INSERT INTO CUSTOMER_TBL VALUES  
( '21', 'MORGANS CANDIES AND TREATS', '5657 W TENTH ST',  
'INDIANAPOLIS', 'IN', '46234', '3172714398', NULL);
```

```
INSERT INTO CUSTOMER_TBL VALUES  
( '43', 'SCHYLERS NOVELTIES', '17 MAPLE ST', 'LEBANON', 'IN',  
'48990', '3174346758', NULL);
```

```
INSERT INTO CUSTOMER_TBL VALUES  
( '287', 'GAVINS PLACE', '9880 ROCKVILLE RD', 'INDIANAPOLIS',  
'IN', '46244', '3172719991', '3172719992');
```

```
INSERT INTO CUSTOMER_TBL VALUES  
( '288', 'HOLLYS GAMEARAMA', '567 US 31 SOUTH', 'WHITELAND',  
'IN', '49980', '3178879023', NULL);
```

```
INSERT INTO CUSTOMER_TBL VALUES
```

```
('590', 'HEATHERS FEATHERS AND THINGS', '4090 N SHADELAND AVE',  
'INDIANAPOLIS', 'IN', '43278', '3175456768', NULL);
```

```
INSERT INTO CUSTOMER_TBL VALUES  
( '610', 'REGANS HOBBIES INC', '451 GREEN ST', 'PLAINFIELD', 'IN',  
'46818', '3178393441', '3178399090');
```

```
INSERT INTO CUSTOMER_TBL VALUES  
( '560', 'ANDYS CANDIES', 'RR 1 BOX 34', 'NASHVILLE', 'IN',  
'48756', '8123239871', NULL);
```

```
INSERT INTO CUSTOMER_TBL VALUES  
( '221', 'RYANS STUFF', '2337 S SHELBY ST', 'INDIANAPOLIS', 'IN',  
'47834', '3175634402', NULL);
```

### E.1.4 ORDERS\_TBL

```
INSERT INTO ORDERS_TBL VALUES  
( '56A901', '232', '11235', '1', '2009-10-22');
```

```
INSERT INTO ORDERS_TBL VALUES  
( '56A917', '12', '907', '100', '2009-09-30');
```

```
INSERT INTO ORDERS_TBL VALUES  
( '32A132', '43', '222', '25', '2009-10-10');
```

```
INSERT INTO ORDERS_TBL VALUES  
( '16C17', '090', '222', '2', '2009-10-17');
```

```
INSERT INTO ORDERS_TBL VALUES  
( '18D778', '287', '90', '10', '2009-10-17');
```

```
INSERT INTO ORDERS_TBL VALUES  
( '23E934', '432', '13', '20', '2009-10-15');
```

### E.1.5 PRODUCTS\_TBL

```
INSERT INTO PRODUCTS_TBL VALUES
('11235', 'WITCH COSTUME', '29.99');

INSERT INTO PRODUCTS_TBL VALUES
('222', 'PLASTIC PUMPKIN 18 INCH', '7.75');

INSERT INTO PRODUCTS_TBL VALUES
('13', 'FALSE PARAFFIN TEETH', '1.10');

INSERT INTO PRODUCTS_TBL VALUES
('90', 'LIGHTED LANTERNS', '14.50');

INSERT INTO PRODUCTS_TBL VALUES
('15', 'ASSORTED COSTUMES', '10.00');

INSERT INTO PRODUCTS_TBL VALUES
('9', 'CANDY CORN', '1.35');

INSERT INTO PRODUCTS_TBL VALUES

('6', 'PUMPKIN CANDY', '1.45');

INSERT INTO PRODUCTS_TBL VALUES
('87', 'PLASTIC SPIDERS', '1.05');

INSERT INTO PRODUCTS_TBL VALUES
('119', 'ASSORTED MASKS', '4.95');
```

## E.2 Oracle

### E.2.1 EMPLOYEE\_TBL



```

INSERT INTO EMPLOYEE_TBL VALUES
('311549902', 'STEPHENS', 'TINA', 'DAWN', 'RR 3 BOX 17A', 'GREENWOOD',
'IN', '47890', '3178784465', NULL);

INSERT INTO EMPLOYEE_TBL VALUES
('442346889', 'PLEW', 'LINDA', 'CAROL', '3301 BEACON', 'INDIANAPOLIS',
'IN', '46224', '3172978990', NULL);

INSERT INTO EMPLOYEE_TBL VALUES
('213764555', 'GLASS', 'BRANDON', 'SCOTT', '1710 MAIN ST', 'WHITELAND',
'IN', '47885', '3178984321', '3175709980');

INSERT INTO EMPLOYEE_TBL VALUES
('313782439', 'GLASS', 'JACOB', NULL, '3789 WHITE RIVER BLVD',
'INDIANAPOLIS', 'IN', '45734', '3175457676', '8887345678');

INSERT INTO EMPLOYEE_TBL VALUES
('220984332', 'WALLACE', 'MARIAH', NULL, '7889 KEYSTONE AVE',
'INDIANAPOLIS', 'IN', '46741', '3173325986', NULL);

INSERT INTO EMPLOYEE_TBL VALUES
('443679012', 'SPURGEON', 'TIFFANY', NULL, '5 GEORGE COURT',
'INDIANAPOLIS', 'IN', '46234', '3175679007', NULL);

```

## E.2.2 EMPLOYEE\_PAY\_TBL

```

INSERT INTO EMPLOYEE_PAY_TBL VALUES
('311549902', 'MARKETING', TO_DATE('1999-05-23', 'YYYY-MM-DD'), NULL, TO_DATE('2009-05-01', 'YYYY-MM-DD'), '40000', NULL);

INSERT INTO EMPLOYEE_PAY_TBL VALUES
('442346889', 'TEAM LEADER', TO_DATE('2000-06-17', 'YYYY-MM-DD'), '14.75', TO_DATE('2009-06-01', 'YYYY-MM-DD'), NULL, NULL);

INSERT INTO EMPLOYEE_PAY_TBL VALUES
('213764555', 'SALES MANAGER', TO_DATE('2004-08-14', 'YYYY-MM-DD'), NULL, TO_DATE('2009-08-01', 'YYYY-MM-DD'), '30000', '2000');

INSERT INTO EMPLOYEE_PAY_TBL VALUES
('313782439', 'SALESMAN', TO_DATE('2007-06-28', 'YYYY-MM-DD'), NULL, NULL, '20000', '1000');

INSERT INTO EMPLOYEE_PAY_TBL VALUES
('220984332', 'SHIPPER', TO_DATE('2006-07-22', 'YYYY-MM-DD'), '11.00', '2009-07-01', NULL, NULL);

INSERT INTO EMPLOYEE_PAY_TBL VALUES
('443679012', 'SHIPPER', TO_DATE('2001-01-14', 'YYYY-MM-DD'), '15.00', '2009-01-01', NULL, NULL);

```

### E.2.3 CUSTOMER\_TBL



```

INSERT INTO CUSTOMER_TBL VALUES
('232', 'LESLIE GLEASON', '798 HARDAWAY DR', 'INDIANAPOLIS',
'IN', '47856', '3175457690', NULL);

INSERT INTO CUSTOMER_TBL VALUES
('109', 'NANCY BUNKER', 'APT A 4556 WATERWAY', 'BROAD RIPPLE',
'IN', '47950', '3174262323', NULL);

INSERT INTO CUSTOMER_TBL VALUES
('345', 'ANGELA DOBK0', 'RR3 BOX 76', 'LEBANON', 'IN', '49967',
'7658970090', NULL);

INSERT INTO CUSTOMER_TBL VALUES
('090', 'WENDY WOLF', '3345 GATEWAY DR', 'INDIANAPOLIS', 'IN',
'46224', '3172913421', NULL);

INSERT INTO CUSTOMER_TBL VALUES
('12', 'MARYS GIFT SHOP', '435 MAIN ST', 'DANVILLE', 'IL', '47978',
'3178567221', '3178523434');

INSERT INTO CUSTOMER_TBL VALUES
('432', 'SCOTTYS MARKET', 'RR2 BOX 173', 'BROWNSBURG', 'IN',
'45687', '3178529835', '3178529836');

INSERT INTO CUSTOMER_TBL VALUES
('333', 'JASONS AND DALLAS GOODIES', 'LAFAYETTE SQ MALL',
'INDIANAPOLIS', 'IN', '46222', '3172978886', '3172978887');

INSERT INTO CUSTOMER_TBL VALUES
('21', 'MORGANS CANDIES AND TREATS', '5657 W TENTH ST',
'INDIANAPOLIS', 'IN', '46234', '3172714398', NULL);

INSERT INTO CUSTOMER_TBL VALUES
('43', 'SCHYLERS NOVELTIES', '17 MAPLE ST', 'LEBANON', 'IN',
'48990', '3174346758', NULL);

INSERT INTO CUSTOMER_TBL VALUES
('287', 'GAVINS PLACE', '9880 ROCKVILLE RD', 'INDIANAPOLIS',
'IN', '46244', '3172719991', '3172719992');

INSERT INTO CUSTOMER_TBL VALUES
('288', 'HOLLYS GAMEARAMA', '567 US 31 SOUTH', 'WHITELAND',
'IN', '49980', '3178879023', NULL);

INSERT INTO CUSTOMER_TBL VALUES
('590', 'HEATHERS FEATHERS AND THINGS', '4090 N SHADELAND AVE',
'INDIANAPOLIS', 'IN', '43278', '3175456768', NULL);

INSERT INTO CUSTOMER_TBL VALUES
('610', 'REGANS HOBBIES INC', '451 GREEN ST', 'PLAINFIELD', 'IN',
'46818', '3178393441', '3178399090');

INSERT INTO CUSTOMER_TBL VALUES
('560', 'ANDYS CANDIES', 'RR 1 BOX 34', 'NASHVILLE', 'IN',
'48756', '8123239871', NULL);

```

```
INSERT INTO CUSTOMER_TBL VALUES
('221', 'RYANS STUFF', '2337 S SHELBY ST', 'INDIANAPOLIS', 'IN',
'47834', '3175634402', NULL);
```

### E.2.4 ORDERS\_TBL

```
INSERT INTO ORDERS_TBL VALUES
('56A901', '232', '11235', '1', TO_DATE('2009-10-22', 'YYYY-MM-DD'));
```

```
INSERT INTO ORDERS_TBL VALUES
('56A917', '12', '907', '100', TO_DATE('2009-09-30', 'YYYY-MM-DD'));
```

```
INSERT INTO ORDERS_TBL VALUES
('32A132', '43', '222', '25', TO_DATE('2009-10-10', 'YYYY-MM-DD'));
```

```
INSERT INTO ORDERS_TBL VALUES
('16C17', '090', '222', '2', TO_DATE('2009-10-17', 'YYYY-MM-DD'));
```

```
INSERT INTO ORDERS_TBL VALUES
('18D778', '287', '90', '10', TO_DATE('2009-10-17', 'YYYY-MM-DD'));
```

```
INSERT INTO ORDERS_TBL VALUES
('23E934', '432', '13', '20', TO_DATE('2009-10-15', 'YYYY-MM-DD'));
```

### E.2.5 PRODUCTS\_TBL

```

INSERT INTO PRODUCTS_TBL VALUES
('11235', 'WITCH COSTUME', '29.99');

INSERT INTO PRODUCTS_TBL VALUES
('222', 'PLASTIC PUMPKIN 18 INCH', '7.75');

INSERT INTO PRODUCTS_TBL VALUES
('13', 'FALSE PARAFFIN TEETH', '1.10');

INSERT INTO PRODUCTS_TBL VALUES
('90', 'LIGHTED LANTERNS', '14.50');

INSERT INTO PRODUCTS_TBL VALUES
('15', 'ASSORTED COSTUMES', '10.00');

INSERT INTO PRODUCTS_TBL VALUES
('9', 'CANDY CORN', '1.35');

INSERT INTO PRODUCTS_TBL VALUES
('6', 'PUMPKIN CANDY', '1.45');

INSERT INTO PRODUCTS_TBL VALUES
('87', 'PLASTIC SPIDERS', '1.05');

INSERT INTO PRODUCTS_TBL VALUES
('119', 'ASSORTED MASKS', '4.95');

```

## [附录F 额外练习](#)

这个附录包含一些额外的练习，而且是针对MySQL的。在这些练习里，首先是说明或问题，然后是需要你在mysql>提示符下输入的符合MySQL语法的SQL代码。请记住，SQL代码在不同的实现中是不同的，所以需要根据所使用的系统来对代码进行调整。请仔细学习这些问题、代码和结果，从而更好地掌握SQL。

1. 新建一个名为BONUS的数据库来进行以下的练习。

```
CREATE DATABASE BONUS;
```

2. 指向新建的数据库。

```
USE BONUS;
```

3. 创建一个表格来记录篮球队。

```
CREATE TABLE TEAMS
( TEAM_ID          INTEGER(2)    NOT NULL,
  NAME             VARCHAR(20)   NOT NULL );
```

4. 创建一个表格记录队员。

```
CREATE TABLE PLAYERS
( PLAYER_ID        INTEGER(2)    NOT NULL,
  LAST             VARCHAR(20)   NOT NULL,
  FIRST           VARCHAR(20)   NOT NULL,
  TEAM_ID          INTEGER(2)    NULL,
  NUMBER           INTEGER(2)    NOT NULL );
```

5. 创建一个表格记录队员的个人信息。

```
CREATE TABLE PLAYER_DATA
( PLAYER_ID        INTEGER(2)    NOT NULL,
  HEIGHT           DECIMAL(4,2)  NOT NULL,
  WEIGHT           DECIMAL(5,2)  NOT NULL );
```

6. 创建一个表格记录比赛。

```
CREATE TABLE GAMES
( GAME_ID          INTEGER(2)    NOT NULL,
  GAME_DT          DATETIME      NOT NULL,

  HOME_TEAM_ID     INTEGER(2)    NOT NULL,
  GUEST_TEAM_ID    INTEGER(3)    NOT NULL );
```

7. 创建一个表格记录每支球队在每场比赛里的成绩。

```
CREATE TABLE SCORES
( GAME_ID      INTEGER(2)      NOT NULL,
  TEAM_ID      INTEGER(2)      NOT NULL,
  SCORE        INTEGER(3)      NOT NULL,
  WIN_LOSE     VARCHAR(4)      NOT NULL );
```

8. 查看创建的这些表。

```
SHOW TABLES;
```

9. 生成篮球队的记录。

```
INSERT INTO TEAMS VALUES ( '1', 'STRING MUSIC' );
INSERT INTO TEAMS VALUES ( '2', 'HACKERS' );
INSERT INTO TEAMS VALUES ( '3', 'SHARP SHOOTERS' );
INSERT INTO TEAMS VALUES ( '4', 'HAMMER TIME' );
```

10. 生成队员的记录。

```
INSERT INTO PLAYERS VALUES ( '1', 'SMITH', 'JOHN', '1', '12' );
INSERT INTO PLAYERS VALUES ( '2', 'BOBBIT', 'BILLY', '1', '2' );
INSERT INTO PLAYERS VALUES ( '3', 'HURTA', 'WIL', '2', '32' );
INSERT INTO PLAYERS VALUES ( '4', 'OUCHY', 'TIM', '2', '22' );
INSERT INTO PLAYERS VALUES ( '5', 'BYRD', 'ERIC', '3', '6' );
INSERT INTO PLAYERS VALUES ( '6', 'JORDAN', 'RYAN', '3', '23' );
INSERT INTO PLAYERS VALUES ( '7', 'HAMMER', 'WALLY', '4', '21' );
INSERT INTO PLAYERS VALUES ( '8', 'HAMMER', 'RON', '4', '44' );
INSERT INTO PLAYERS VALUES ( '11', 'KNOTGOOD', 'AL', NULL, '0' );
```

11. 生成队员的个人信息。

```
INSERT INTO PLAYER_DATA VALUES ('1','71','180');
INSERT INTO PLAYER_DATA VALUES ('2','58','195');
INSERT INTO PLAYER_DATA VALUES ('3','72','200');
INSERT INTO PLAYER_DATA VALUES ('4','74','170');
INSERT INTO PLAYER_DATA VALUES ('5','71','182');
INSERT INTO PLAYER_DATA VALUES ('6','72','289');
INSERT INTO PLAYER_DATA VALUES ('7','79','250');
INSERT INTO PLAYER_DATA VALUES ('8','73','193');
INSERT INTO PLAYER_DATA VALUES ('11','85','310');
```

12. 基于已经安排的比赛生成GAMES表里的记录。

```
INSERT INTO GAMES VALUES ('1','2002-05-01','1','2');
INSERT INTO GAMES VALUES ('2','2002-05-02','3','4');
INSERT INTO GAMES VALUES ('3','2002-05-03','1','3');
INSERT INTO GAMES VALUES ('4','2002-05-05','2','4');
INSERT INTO GAMES VALUES ('5','2002-05-05','1','2');
INSERT INTO GAMES VALUES ('6','2002-05-09','3','4');
INSERT INTO GAMES VALUES ('7','2002-05-10','2','3');
INSERT INTO GAMES VALUES ('8','2002-05-11','1','4');
INSERT INTO GAMES VALUES ('9','2002-05-12','2','3');
INSERT INTO GAMES VALUES ('10','2002-05-15','1','4');
```

13. 基于已经进行的比赛生成SCORES表里的记录。

```
INSERT INTO SCORES VALUES ('1','1','66','LOSE');
INSERT INTO SCORES VALUES ('2','3','78','WIN');
INSERT INTO SCORES VALUES ('3','1','45','LOSE');
INSERT INTO SCORES VALUES ('4','2','56','LOSE');
INSERT INTO SCORES VALUES ('5','1','100','WIN');
INSERT INTO SCORES VALUES ('6','3','67','LOSE');
INSERT INTO SCORES VALUES ('7','2','57','LOSE');
INSERT INTO SCORES VALUES ('8','1','98','WIN');
INSERT INTO SCORES VALUES ('9','2','56','LOSE');
```

```
INSERT INTO SCORES VALUES ('10','1','46','LOSE');
```

```
INSERT INTO SCORES VALUES ('1','2','75','WIN');  
INSERT INTO SCORES VALUES ('2','4','46','LOSE');  
INSERT INTO SCORES VALUES ('3','3','87','WIN');  
INSERT INTO SCORES VALUES ('4','4','99','WIN');  
INSERT INTO SCORES VALUES ('5','2','88','LOSE');  
INSERT INTO SCORES VALUES ('6','4','77','WIN');  
INSERT INTO SCORES VALUES ('7','3','87','WIN');  
INSERT INTO SCORES VALUES ('8','4','56','LOSE');  
INSERT INTO SCORES VALUES ('9','3','87','WIN');  
INSERT INTO SCORES VALUES ('10','4','78','WIN');
```

14. 球员的平均身高是多少？

```
SELECT AVG(HEIGHT) FROM PLAYER_DATA;
```

15. 球员的平均体重是多少？

```
SELECT AVG(WEIGHT) FROM PLAYER_DATA;
```

16. 像下面这样查看球员信息：

```
NAME=LAST NUMBER=N HEIGHT=N WEIGHT=N  
SELECT CONCAT('NAME=',P1.LAST,' NUMBER=',P1.NUMBER,'  
HEIGHT=',P2.HEIGHT,' WEIGHT=',P2.WEIGHT)  
FROM PLAYERS P1,  
     PLAYER_DATA P2  
WHERE P1.PLAYER_ID = P2.PLAYER_ID;
```

17. 像下面这样创建一个球队的人名单：

```
TEAM NAME          LAST, FIRST    NUMBER  
SELECT T.NAME, CONCAT(P.LAST,' ',P.FIRST), P.NUMBER  
FROM TEAMS T,  
     PLAYERS P  
WHERE T.TEAM_ID = P.TEAM_ID;
```

18. 哪支球队在全部比赛中获得了最多的分数？

```
SELECT T.NAME, SUM(S.SCORE)
FROM TEAMS T,
      SCORES S
WHERE T.TEAM_ID = S.TEAM_ID
GROUP BY T.NAME
ORDER BY 2 DESC;
```

19. 哪只球队在一场比赛里获得了最高分数？

```
SELECT MAX(SCORE)
FROM SCORES;
```

20. 在一场比赛里，两队分数之和最高是多少？

```
SELECT GAME_ID, SUM(SCORE)
FROM SCORES
GROUP BY GAME_ID
ORDER BY 2 DESC;
```

21. 哪个球员不属于任何球队？

```
SELECT LAST, FIRST, TEAM_ID
FROM PLAYERS
WHERE TEAM_ID IS NULL;
```

22. 一共有多少只球队？

```
SELECT COUNT(*) FROM TEAMS;
```

23. 一共有多少球员？

```
SELECT COUNT(*) FROM PLAYERS;
```

24. 2002年5月5日有多少场比赛？



```
SELECT COUNT(*) FROM GAMES
WHERE GAME_DT = '2002-05-05';
```

25. 谁是最高的球员？

```
SELECT P.LAST, P.FIRST, PD.HEIGHT
FROM PLAYERS P,
     PLAYER_DATA PD
WHERE P.PLAYER_ID = PD.PLAYER_ID
ORDER BY 3 DESC;
OR
SELECT MAX(HEIGHT) FROM PLAYER_DATA;
SELECT P.LAST, P.FIRST, PD.HEIGHT
FROM PLAYERS P,
     PLAYER_DATA PD
WHERE HEIGHT = 85;
```

26. 把Ron Hammer的记录从数据库里删除，用Al Knotgood替换他。

```
SELECT PLAYER_ID
FROM PLAYERS
WHERE LAST = 'HAMMER'
     AND FIRST = 'RON';
DELETE FROM PLAYERS WHERE PLAYER_ID = '8';
DELETE FROM PLAYER_DATA WHERE PLAYER_ID = '8';
SELECT PLAYER_ID
FROM PLAYERS
WHERE LAST = 'KNOTGOOD'
     AND FIRST = 'AL';
UPDATE PLAYERS
SET TEAM_ID = '4'
WHERE PLAYER_ID = '11';
```

27. 谁是Al Knotgood的新队友？

```

SELECT TEAMMATE.LAST, TEAMMATE.FIRST
FROM PLAYERS TEAMMATE,
     PLAYERS P
WHERE P.TEAM_ID = TEAMMATE.TEAM_ID
     AND P.LAST = 'KNOTGOOD'
     AND P.FIRST = 'AL';

```

28. 生成一个列表，列出全部比赛和比赛日期，以及每场比赛的主队和客队。

```

SELECT G.GAME_ID, HT.NAME, GT.NAME
FROM GAMES G,
     TEAMS HT,
     TEAMS GT
WHERE HT.TEAM_ID = G.HOME_TEAM_ID
     AND GT.TEAM_ID = G.GUEST_TEAM_ID;

```

29. 为数据库里的全部姓名设置索引。我们经常会根据姓名进行搜索，所以一般会被设置索引。

```

CREATE INDEX TEAM_IDX
ON TEAMS (NAME);
CREATE INDEX PLAYERS_IDX
ON PLAYERS (LAST, FIRST);

```

30. 哪支球队赢的比赛最多？

```

SELECT T.NAME, COUNT(S.WIN_LOSE)
FROM TEAMS T,
     SCORES S
WHERE T.TEAM_ID = S.TEAM_ID
     AND S.WIN_LOSE = 'WIN'
GROUP BY T.NAME
ORDER BY 2 DESC;

```

31. 哪支球队输的比赛最多？

```

SELECT T.NAME, COUNT(S.WIN_LOSE)
FROM TEAMS T,
     SCORES S
WHERE T.TEAM_ID = S.TEAM_ID
     AND S.WIN_LOSE = 'LOSE'
GROUP BY T.NAME
ORDER BY 2 DESC;

```

32. 哪支球队的场均得分最高？

```

SELECT T.NAME, AVG(S.SCORE)
FROM TEAMS T,
     SCORES S
WHERE T.TEAM_ID = S.TEAM_ID
GROUP BY T.NAME
ORDER BY 2 DESC;

```

33. 生成一个报告来展示每支球队的记录。输出结果首先按赢的场次多排序，再按输的场次少排序。

```

SELECT T.NAME, SUM(REPLACE(S.WIN_LOSE, 'WIN', 1)) WINS,
     SUM(REPLACE(S.WIN_LOSE, 'LOSE', 1)) LOSSES
FROM TEAMS T,
     SCORES S
WHERE T.TEAM_ID = S.TEAM_ID
GROUP BY T.NAME
ORDER BY 2 DESC, 3;

```

34. 每场比赛的最终比分是多少？

```

SELECT G.GAME_ID,
       HOME_TEAMS.NAME "HOME TEAM", HOME_SCORES.SCORE,
       GUEST_TEAMS.NAME "GUEST TEAM", GUEST_SCORES.SCORE
FROM GAMES G,
     TEAMS HOME_TEAMS,
     TEAMS GUEST_TEAMS,
     SCORES HOME_SCORES,
     SCORES GUEST_SCORES
WHERE G.HOME_TEAM_ID = HOME_TEAMS.TEAM_ID
     AND G.GUEST_TEAM_ID = GUEST_TEAMS.TEAM_ID
     AND HOME_SCORES.GAME_ID = G.GAME_ID
     AND GUEST_SCORES.GAME_ID = G.GAME_ID
     AND HOME_SCORES.TEAM_ID = G.HOME_TEAM_ID
     AND GUEST_SCORES.TEAM_ID = G.GUEST_TEAM_ID
     ORDER BY G.GAME_ID;

```

## [术语表](#)

**别名** 表或字段的另一个名称。

**ANSI** 美国国家标准化组织。该组织负责为各种课题发布标准。SQL标准即为该组织所发布。

**应用程序** 一组菜单、表单、报告和代码，通常利用数据库执行商务功能。

**缓存** 内存里的一个区域，用于编辑或执行SQL。

**笛卡尔积** 在WHERE子句里不结合表而产生的结果。当查询里的表没有结合时，一个表里的全部记录都与另一个表里的每条记录配对。

**客户端** 客户端通常是个人计算机，但也可以是一台服务器，它依赖于另一个计算机的数据、服务或处理。客户端程序可以让客户端计算机与服务器通信。

**列** 表的组成部分，具有名称和特定的数据类型。

**COMMIT** 一个命令，让数据的修改生效。

**组合索引** 由两个或多个字段组成的索引。

**条件** 查询的WHERE子句里的搜索准则，其值为TRUE或FALSE。

**常数** 不会变化的值。

**约束** 在数据级对数据的限制。

**光标** 内存里的一个工作区域，使用SQL语句对数据集进行以行为单位的操作。

**数据目录** 系统目录的别称。参见系统目录。

**数据类型** 以不同的类型定义数据，比如数值、日期或字符。

**数据库** 数据的集合，通常用一系列表来组织数据。

**DBA** 数据库管理员，是负责管理数据库的人。

**DDL** 数据定义语言。这部分SQL语句专门用于定义数据库对象，比如表格、视图和函数。

**默认** 不指定任何值时使用的值。

**DISTINCT** 一个选项，在SELECT语句里用于返回不重复的值。

**DML** 数据操作语言。这部分SQL语句专门用于操作数据，比如更新数据。

**域** 一个与数据类型相关联的对象，还可以包含约束；类似于自定义类型。

**DQL** 数据查询语言。这部分SQL语句专门用于利用SELECT语句查询数据。

**终端用户** 根据需要对数据库进行查询或操作数据的用户，是数据库存在的根本原因。

**字段** 表格中列的别称，参见“列”。

**外键** 一个或多个字段，其值基于另一个表的主键。

**全表扫描** 在不使用索引的情况下，查询对表进行的搜索。

**函数** 预定义的操作，可以用在SQL语句里操作数据。

**GUI** 图形用户界面。当应用接口需要向用户提供图形元素以便进

行交互的时候，往往需要使用GUI。

**主机** 数据库所在的计算机。

**索引** 指向表格数据的指针，能够提高表格访问的效率。

**JDBC** Java数据库连接软件。允许Java程序与数据库进行通信来处理数据。

**结合** 通过链接字段组合来自不同表格的数据。用在SQL语句的WHERE子句里。

**键** 一个或多个字段，用于区别表格里的不同记录。

**规格化** 在设计数据库时，把大型表格划分为较小的、更容易管理的表格，从而减少冗余。

**NULL值** 一个未知值。

**对象** 数据库里的元素，比如触发器、表格、视图和过程。

**ODBC** 开放数据库连接，是与数据库进行标准通信的软件。ODBC通常用于不同实现之间的数据库通信，以及客户端程序与数据库的通信。

**操作符** 用于执行操作的保留字或符号。

**优化器** 数据库的内部机制，决定如何执行SQL语句和返回结果。

**参数** 用于解析SQL语句或程序的一个值或一个范围内的值。

**主键** 表格里一个专用字段，用于区别不同的记录。

**权限** 授予用户的特定许可，允许在数据库里执行特定操作。

**过程** 一组被保存起来的指令，可以重复调用和执行。

**PUBLIC** 数据库的一个用户账户，代表数据库的全部用户。

**查询** 用于从数据库检索数据的SQL语句。

**记录** 表格里一行数据的别称，参见“行”。

**引用完整性** 确保来自一个字段的值依赖于另一个字段的值，用于确保数据库中数据的一致性。它通常用于两个表之间，但有时也可以用于一个表，让表来引用自己。自引用表格被称为递归关系。在数据库

中，通常称之为外键关系。

**关系型数据库** 由表格组成的数据库，表格由记录组成，这些记录具有相同的数据元素，而这些表格之间通过共同的字段产生关联。

**角色** 与一组系统权限和/或对象权限相关联的数据库对象，用于简化安全管理工作。

**ROLLBACK** 一个命令，可以撤销自最后一个COMMIT或SAVEPOINT命令之后的全部事务。

**行** 表里一组数据。

**保存点** 事务里的指定点，用于回退或撤销修改。

**规划** 一个用户所属的一组相关联的数据库对象。

**安全** 确保数据库里的数据受到全时全方位保护的过程。

**SQL** 结构化查询语言。专为数据库设计，用于在数据库中进行操作。

**存储过程** 存储在数据库里的、可以直接执行的SQL代码。

**子查询** 嵌套在另一个SQL语句里的SELECT语句。

**异名** 赋予表格或视图的另一个名称。

**SQL语法** 规定SQL语句结构中必要部分和可选部分的一组规则。

**系统目录** 包含数据库相关信息的表格与视图的集合。

**表格** 关系型数据库里数据的基本逻辑存储单元。

**事务** 以一个整体执行的一个或多个SQL语句。

**触发器** 根据数据库里特定事件运行的存储过程，比如在表格更新之前或之后。

**自定义类型** 由用户定义的数据类型，可以用于定义表格字段。

**变量** 能够变化的值。

**视图** 基于一个或多个表格创建的数据库对象，能够像表格一样被使用。视图是一个虚拟表格，不需要存储数据的空间。

## 献词

这本书要献给我的父母Thomas和Karlyn Stephens，你们总是鼓励我能够实现目标。本书还要献给我聪明的儿子Daniel和漂亮的女儿Autunn和Alivia，希望你们在达到梦想之前永远不要停下前进的脚步。

——Ryan

这本书要献给我的家人：我妻子Linda，我母亲Betty，我的孩子Leslie、Nancy、Angela和Wendy，我的孙子Andy、Ryan、Holly、Morgan、Schyler、Heather、Gavin、Regan、Caleigh和Cameron，我的女婿Jason和Dilas。感谢你们对我工作的支持，我爱你们。

——Poppy

我要把本书献给我的妻子Jackie，感谢她在本书编写期间对我的理解与支持。

——Arie

## 致谢

感谢在本书各个版本编写过程中对我们给予理解的所有人，特别是我们的妻子 Tina 和Linda。感谢Arie Jones加入我们的工作并给予了巨大的帮助。感谢Sams出版社的编辑团队，你们的辛勤工作让这个版本比以前更加出色，与你们共事非常愉快。

——Ryan和Ron



图书在版编目（CIP）数据

SQL入门经典:第5版/（美）斯蒂芬森（Stephens,R.）,（美）普劳（Plew,R.）,(美)琼斯（Jones,A.D.）着；井中月,郝记生译.--北京:人民邮电出版社,2011.11

ISBN 978-7-115-26407-7

I.①S... II.①斯...②普...③琼...④井...⑤郝... III.①关系数据库—数据库管理系统 IV.①TP311.138

中国版本图书馆CIP数据核字（2011）第189007号

版权声明

**Ryan Stephens, Ron Plew, Arie D. Jones: Sams Teach Yourself SQL in 24 Hours (Fifth Edition)**

**ISBN: 0672335417**

**Copyright © 2011 by Sams Publishing.**

**Authorized translation from the English languages edition published by Sams.**

**All rights reserved.**

本书中文简体字版由美国**Sams**出版公司授权人民邮电出版社出版。未经出版者书面许可，对本书任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

**SQL入门经典（第5版）**

◆着 [美]Ryan Stephens Ron Plew Arie D.Jones

译 井中月 郝记生

责任编辑 傅道坤

◆人民邮电出版社出版发行 北京市崇文区夕照寺街14号

邮编 100061 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

北京艺辉印刷有限公司印刷

◆开本：787×1092 1/16

印张：22.75

字数：562千字 2011年11月第1版

印数：1-3500册 2011年11月北京第1次印刷

著作权合同登记号 图字：01-2011-4655号

ISBN 978-7-115-26407-7

定价：45.00元

读者服务热线：(010)67132705 印装质量热线：(010)67129223

反盗版热线：(010)67171154